W3School 后端教程合集

wizardforcel

Published with GitBook

目錄

W3School 后端教程合集

来源：菜鸟教程

整理：飞龙

感谢菜鸟教程站长的翻译和奉献。

W3School PHP教程

来源：PHP教程

整理：飞龙

PHP 基础

PHP 简介

PHP 脚本在服务器上执行。

您应当具备的基础知识

在继续学习之前，您需要对下面的知识有基本的了解：

- HTML
- CSS
- JavaScript

如果您希望首先学习这些项目，请在我们的 首页 访问这些教程。

什么是 PHP？

- PHP 是 "PHP Hypertext Preprocessor" 的首字母缩略词
- PHP 是一种被广泛使用的开源脚本语言
- PHP 脚本在服务器上执行
- PHP 没有成本，可供免费下载和使用

PHP 是一门令人惊叹的流行语言！

- 它强大到足以成为在网络上最大的博客系统的核心（WordPress）！
- 它深邃到足以运行最大的社交网络（facebook）！
- 而它的易用程度足以成为初学者的首选服务器端语言！

什么是 PHP 文件？

- PHP 文件能够包含文本、HTML、CSS 以及 PHP 代码
- PHP 代码在服务器上执行，而结果以纯文本返回浏览器
- PHP 文件的后缀是 ".php"

PHP 能够做什么？

- PHP 能够生成动态页面内容
- PHP 能够创建、打开、读取、写入、删除以及关闭服务器上的文件

- PHP 能够接收表单数据
- PHP 能够发送并取回 cookies
- PHP 能够添加、删除、修改数据库中的数据
- PHP 能够限制用户访问网站中的某些页面
- PHP 能够对数据进行加密

通过 PHP，您可以不受限于只输出 HTML。您还能够输出图像、PDF 文件、甚至 Flash 影片。您也可以输出任何文本，比如 XHTML 和 XML。

为什么使用 PHP？

- PHP 运行于各种平台（Windows, Linux, Unix, Mac OS X 等等）
- PHP 兼容几乎所有服务器（Apache, IIS 等等）
- PHP 支持多种数据库
- PHP 是免费的。请从官方 PHP 资源下载：www.php.net
- PHP 易于学习，并可高效地运行在服务器端

PHP 安装

我需要什么？

如需开始使用 PHP，您可以：

- 使用支持 PHP 和 MySQL 的 web 主机
- 在您的 PC 上安装 web 服务器，然后安装 PHP 和 MySQL。

使用支持 PHP 的 Web 主机

如果您的服务器支持 PHP，那么您无需做任何事情。

只要创建 .php 文件，然后上传到 web 目录中即可。服务器会自动对它们进行解析。

您无需编译或安装任何额外的工具。

因为 PHP 是免费的，大多数 web 主机都支持 PHP。

在您的 PC 上运行 PHP

不过如果您的服务器不支持 PHP，那么您必须：

- 安装 web 服务器
- 安装 PHP
- 安装数据库，比如 MySQL

官方的 PHP 网站 (PHP.net) 提供了 PHP 的安装说明：http://php.net/manual/zh/install.php

注释

提示：如需在 Windows 平台设置并立即运行 PHP，您还可以：

下载 WebMatrix

PHP 语法

PHP 脚本在服务器上执行，然后向浏览器发送回纯 **HTML** 结果。

基础 PHP 语法

PHP 脚本可放置于文档中的任何位置。

PHP 脚本以 *<?php* 开头，以 *?>* 结尾：

```
<?php
// 此处是 PHP 代码
?>
```

PHP 文件的默认文件扩展名是 ".php"。

PHP 文件通常包含 HTML 标签以及一些 PHP 脚本代码。

下面的例子是一个简单的 PHP 文件，其中包含了使用内建 PHP 函数 "echo" 在网页上输出文本 "Hello World!" 的一段 PHP 脚本：

实例

```
<!DOCTYPE html>
<html>
<body>

<h1>我的第一张 PHP 页面</h1>

<?php
echo "Hello World!";
?>

</body>
</html>
```

注释：PHP 语句以分号结尾（;）。PHP 代码块的关闭标签也会自动表明分号（因此在 PHP 代码块的最后一行不必使用分号）。

PHP 中的注释

PHP 代码中的注释不会被作为程序来读取和执行。它唯一的作用是供代码编辑者阅读。

注释用于：

- 使其他人理解您正在做的工作 - 注释可以让其他程序员了解您在每个步骤进行的工作（如果您供职于团队）
- 提醒自己做过什么 - 大多数程序员都曾经历过一两年后对项目进行返工，然后不得不重新考虑他们做过的事情。注释可以记录您在写代码时的思路。

PHP 支持三种注释：

实例

```
<!DOCTYPE html>
<html>
<body>

<?php
// 这是单行注释

# 这也是单行注释

/*
这是多行注释块
它横跨了
多行
*/
?>

</body>
</html>
```

PHP 大小写敏感

在 PHP 中，所有用户定义的函数、类和关键词（例如 if、else、echo 等等）都对大小写不敏感。

在下面的例子中，所有这三天 echo 语句都是合法的（等价）：

实例

```
<!DOCTYPE html>
<html>
<body>

<?php
ECHO "Hello World!<br>";
echo "Hello World!<br>";
EcHo "Hello World!<br>";
?>

</body>
</html>
```

不过在 PHP 中，所有变量都对大小写敏感。

在下面的例子中，只有第一条语句会显示 $color 变量的值（这是因为 $color、$COLOR 以及 $coLOR 被视作三个不同的变量）：

实例

```
<!DOCTYPE html>
<html>
<body>

<?php
$color="red";
echo "My car is " . $color . "<br>";
echo "My house is " . $COLOR . "<br>";
echo "My boat is " . $coLOR . "<br>";
?>

</body>
</html>
```

PHP 变量

变量是存储信息的容器：

实例

```
<?php
$x=5;
$y=6;
$z=$x+$y;
echo $z;
?>
```

类似代数

```
x=5
y=6
z=x+y
```

在代数中我们使用字母（比如 x）来保存值（比如 5）。

从上面的表达式 z=x+y，我们能够计算出 z 的值是 11。

在 PHP 中，这三个字母被称为变量。

注释：请把变量视为存储数据的容器。

PHP 变量

正如代数，PHP 变量可用于保存值（x=5）和表达式（z=x+y）。

变量的名称可以很短（比如 x 和 y），也可以取更具描述性的名称（比如 carname、total_volume）。

PHP 变量规则：

- 变量以 $ 符号开头，其后是变量的名称
- 变量名称必须以字母或下划线开头
- 变量名称不能以数字开头
- 变量名称只能包含字母数字字符和下划线（A-z、0-9 以及 _）
- 变量名称对大小写敏感（$y 与 $Y 是两个不同的变量）

注释：PHP 变量名称对大小写敏感！

创建 PHP 变量

PHP 没有创建变量的命令。

变量会在首次为其赋值时被创建：

实例

```
<?php
$txt="Hello world!";
$x=5;
$y=10.5;
?>
```

以上语句执行后，变量 txt 会保存值 Hello world!，变量 x 会保存值 5，变量 y 会保存值 10.5。

注释：如果您为变量赋的值是文本，请用引号包围该值。

PHP 是一门类型松散的语言

在上面的例子中，请注意我们不必告知 PHP 变量的数据类型。

PHP 根据它的值，自动把变量转换为正确的数据类型。

在诸如 C 和 C++ 以及 Java 之类的语言中，程序员必须在使用变量之前声明它的名称和类型。

PHP 变量作用域

在 PHP 中，可以在脚本的任意位置对变量进行声明。

变量的作用域指的是变量能够被引用/使用的那部分脚本。

PHP 有三种不同的变量作用域：

- local（局部）
- global（全局）
- static（静态）

Local 和 Global 作用域

函数之外声明的变量拥有 Global 作用域，只能在函数以外进行访问。

函数内部声明的变量拥有 LOCAL 作用域，只能在函数内部进行访问。

下面的例子测试了带有局部和全局作用域的变量：

实例

```
<?php
$x=5; // 全局作用域

function myTest() {
  $y=10; // 局部作用域
  echo "<p>测试函数内部的变量：</p>";
  echo "变量 x 是：$x";
  echo "<br>";
  echo "变量 y 是：$x";
}

myTest();

echo "<p>测试函数之外的变量：</p>";
echo "变量 x 是：$x";
echo "<br>";
echo "变量 y 是：$x";
?>
```

在上例中，有两个变量 $x 和 $y，以及一个函数 myTest()。$x 是全局变量，因为它是在函数之外声明的，而 $y 是局部变量，因为它是在函数内声明的。

如果我们在 myTest() 函数内部输出两个变量的值，$y 会输出在本地声明的值，但是无法 $x 的值，因为它在函数之外创建。

然后，如果在 myTest() 函数之外输出两个变量的值，那么会输出 $x 的值，但是不会输出 $y 的值，因为它是局部变量，并且在 myTest() 内部创建。

注释：您可以在不同的函数中创建名称相同的局部变量，因为局部变量只能被在其中创建它的函数识别。

PHP global 关键词

global 关键词用于访问函数内的全局变量。

要做到这一点，请在（函数内部）变量前面使用 global 关键词：

实例

```
<?php
$x=5;
$y=10;

function myTest() {
  global $x,$y;
  $y=$x+$y;
}

myTest();
echo $y; // 输出 15
?>
```

PHP 同时在名为 $GLOBALS[index] 的数组中存储了所有的全局变量。下标存有变量名。这个数组在函数内也可以访问，并能够用于直接更新全局变量。

上面的例子可以这样重写：

实例

```
<?php
$x=5;
$y=10;

function myTest() {
  $GLOBALS['y']=$GLOBALS['x']+$GLOBALS['y'];
}

myTest();
echo $y; // 输出 15
?>
```

PHP static 关键词

通常，当函数完成/执行后，会删除所有变量。不过，有时我需要不删除某个局部变量。实现这一点需要更进一步的工作。

要完成这一点，请在您首次声明变量时使用 *static* 关键词：

实例

```
<?php

function myTest() {
  static $x=0;
  echo $x;
  $x++;
}

myTest();
myTest();
myTest();

?>
```

然后，每当函数被调用时，这个变量所存储的信息都是函数最后一次被调用时所包含的信息。

注释：该变量仍然是函数的局部变量。

PHP 5 echo 和 print 语句

在 **PHP** 中，有两种基本的输出方法：**echo** 和 **print**。

在本教程中，我们几乎在每个例子中都会用到 echo 和 print。因此，本节为您讲解更多关于这两条输出语句的知识。

PHP echo 和 print 语句

echo 和 print 之间的差异：

- echo - 能够输出一个以上的字符串
- print - 只能输出一个字符串，并始终返回 1

提示：echo 比 print 稍快，因为它不返回任何值。

PHP echo 语句

echo 是一个语言结构，有无括号均可使用：echo 或 echo()。

显示字符串

下面的例子展示如何用 echo 命令来显示不同的字符串（同时请注意字符串中能包含 HTML 标记）：

```
<?php
echo "<h2>PHP is fun!</h2>";
echo "Hello world!<br>";
echo "I'm about to learn PHP!<br>";
echo "This", " string", " was", " made", " with multiple parameters.";
?>
```

显示变量

下面的例子展示如何用 echo 命令来显示字符串和变量：

```
<?php
$txt1="Learn PHP";
$txt2="W3School.com.cn";
$cars=array("Volvo","BMW","SAAB");

echo $txt1;
echo "<br>";
echo "Study PHP at $txt2";
echo "My car is a {$cars[0]}";
?>
```

PHP print 语句

print 也是语言结构，有无括号均可使用：print 或 print()。

显示字符串

下面的例子展示如何用 print 命令来显示不同的字符串（同时请注意字符串中能包含 HTML 标记）：

```
<?php
print "<h2>PHP is fun!</h2>";
print "Hello world!<br>";
print "I'm about to learn PHP!";
?>
```

显示变量

下面的例子展示如何用 print 命令来显示字符串和变量：

```
<?php
$txt1="Learn PHP";
$txt2="W3School.com.cn";
$cars=array("Volvo","BMW","SAAB");

print $txt1;
print "<br>";
print "Study PHP at $txt2";
print "My car is a {$cars[0]}";
?>
```

PHP 数据类型

字符串、整数、浮点数、逻辑、数组、对象、**NULL**。

PHP 字符串

字符串是字符序列，比如 "Hello world!"。

字符串可以是引号内的任何文本。您可以使用单引号或双引号：

实例

```
<?php
$x = "Hello world!";
echo $x;
echo "<br>";
$x = 'Hello world!';
echo $x;
?>
```

PHP 整数

整数是没有小数的数字。

整数规则：

- 整数必须有至少一个数字（0-9）
- 整数不能包含逗号或空格
- 整数不能有小数点
- 整数正负均可
- 可以用三种格式规定整数：十进制、十六进制（前缀是 0x）或八进制（前缀是 0）

在下面的例子中，我们将测试不同的数字。PHP var_dump() 会返回变量的数据类型和值：

实例

```
<?php
$x = 5985;
var_dump($x);
echo "<br>";
$x = -345; // 负数
var_dump($x);
echo "<br>";
$x = 0x8C; // 十六进制数
var_dump($x);
echo "<br>";
$x = 047; // 八进制数
var_dump($x);
?>
```

PHP 浮点数

浮点数是有小数点或指数形式的数字。

在下面的例子中，我们将测试不同的数字。PHP var_dump() 会返回变量的数据类型和值：

实例

```
<?php
$x = 10.365;
var_dump($x);
echo "<br>";
$x = 2.4e3;
var_dump($x);
echo "<br>";
$x = 8E-5;
var_dump($x);
?>
```

PHP 逻辑

逻辑是 true 或 false。

```
$x=true;
$y=false;
```

逻辑常用于条件测试。您将在本教程稍后的章节学到更多有关条件测试的知识。

PHP 数组

数组在一个变量中存储多个值。

在下面的例子中，我们将测试不同的数组。PHP var_dump() 会返回变量的数据类型和值：

实例

```
<?php
$cars=array("Volvo","BMW","SAAB");
var_dump($cars);
?>
```

您将在本教程稍后的章节学到更多有关数组的知识。

PHP 对象

对象是存储数据和有关如何处理数据的信息的数据类型。

在 PHP 中，必须明确地声明对象。

首先我们必须声明对象的类。对此，我们使用 class 关键词。类是包含属性和方法的结构。

然后我们在对象类中定义数据类型，然后在该类的实例中使用此数据类型：

实例

```
<?php
class Car
{
  var $color;
  function Car($color="green") {
    $this->color = $color;
  }
  function what_color() {
    return $this->color;
  }
}
?>
```

您将在本教程稍后的章节学到更多有关对象的知识。

PHP NULL 值

特殊的 NULL 值表示变量无值。NULL 是数据类型 NULL 唯一可能的值。

NULL 值标示变量是否为空。也用于区分空字符串与空值数据库。

可以通过把值设置为 NULL，将变量清空：

实例

```
<?php
$x="Hello world!";
$x=null;
var_dump($x);
?>
```

PHP 字符串函数

字符串是字符序列，比如 **"Hello world!"**。

PHP 字符串函数

在本节中，我们将学习常用的字符串操作函数。

PHP strlen() 函数

strlen() 函数返回字符串的长度，以字符计。

下例返回字符串 "Hello world!" 的长度：

实例

```
<?php
echo strlen("Hello world!");
?>
```

以上代码的输出是：12

提示：strlen() 常用于循环和其他函数，在确定字符串何时结束很重要时。（例如，在循环中，我们也许需要在字符串的最后一个字符之后停止循环）。

PHP strpos() 函数

strpos() 函数用于检索字符串内指定的字符或文本。

如果找到匹配，则会返回首个匹配的字符位置。如果未找到匹配，则将返回 FALSE。

下例检索字符串 "Hello world!" 中的文本 "world"：

实例

```
<?php
echo strpos("Hello world!","world");
?>
```

以上代码的输出是：6。

提示：上例中字符串 "world" 的位置是 6。是 6（而不是 7）的理由是，字符串中首字符的位置是 0 而不是 1。

完整的 PHP String 参考手册

如需完整的字符串函数参考手册，请访问我们的 PHP String 参考手册。

这个手册提供了每个函数的简要描述和实例！

PHP 常量

常量类似变量，但是常量一旦被定义就无法更改或撤销定义。

PHP 常量

常量是单个值的标识符（名称）。在脚本中无法改变该值。

有效的常量名以字符或下划线开头（常量名称前面没有 $ 符号）。

注释：与变量不同，常量贯穿整个脚本是自动全局的。

设置 PHP 常量

如需设置常量，请使用 define() 函数 - 它使用三个参数：

1. 首个参数定义常量的名称
2. 第二个参数定义常量的值
3. 可选的第三个参数规定常量名是否对大小写敏感。默认是 false。

下例创建了一个对大小写敏感的常量，值为 "Welcome to W3School.com.cn!"：

实例

```
<?php
define("GREETING", "Welcome to W3School.com.cn!");
echo GREETING;
?>
```

下例创建了一个对大小写不敏感的常量，值为 "Welcome to W3School.com.cn!"：

实例

```
<?php
define("GREETING", "Welcome to W3School.com.cn!", true);
echo greeting;
?>
```

PHP 运算符

本节展示了可用于 **PHP** 脚本中的各种运算符.

PHP 算数运算符

运算符	名称	例子	结果
+	加法	$x + $y	$x 与 $y 求和
-	减法	$x - $y	$x 与 $y 的差数
*	乘法	$x * $y	$x 与 $y 的乘积
/	除法	$x / $y	$x 与 $y 的商数
%	模数	$x % $y	$x 除 $y 的余数

下例展示了使用不同算数运算符的不同结果：

实例

```
<?php
$x=10;
$y=6;
echo ($x + $y); // 输出 16
echo ($x - $y); // 输出 4
echo ($x * $y); // 输出 60
echo ($x / $y); // 输出 1.6666666666667
echo ($x % $y); // 输出 4
?>
```

PHP 赋值运算符

PHP 赋值运算符用于向变量写值。

PHP 中基础的赋值运算符是 "="。这意味着右侧赋值表达式会为左侧运算数设置值。

赋值	等同于	描述
x = y	x = y	右侧表达式为左侧运算数设置值。
x += y	x = x + y	加
x -= y	x = x - y	减
x *= y	x = x * y	乘
x /= y	x = x / y	除
x %= y	x = x % y	模数

下例展示了使用不同赋值运算符的不同结果：

实例

```
<?php
$x=10;
echo $x; // 输出 10

$y=20;
$y += 100;
echo $y; // 输出 120

$z=50;
$z -= 25;
echo $z; // 输出 25

$i=5;
$i *= 6;
echo $i; // 输出 30

$j=10;
$j /= 5;
echo $j; // 输出 2

$k=15;
$k %= 4;
echo $k; // 输出 3
?>
```

PHP 字符串运算符

运算符	名称	例子	结果
.	串接	$txt1 = "Hello" $txt2 = $txt1 . " world!"	现在 $txt2 包含 "Hello world!"
.=	串接赋值	$txt1 = "Hello" $txt1 .= " world!"	现在 $txt1 包含 "Hello world!"

下例展示了使用字符串运算符的结果：

实例

```
<?php
$a = "Hello";
$b = $a . " world!";
echo $b; // 输出 Hello world!

$x="Hello";
$x .= " world!";
echo $x; // 输出 Hello world!
?>
```

PHP 递增/递减运算符

运算符	名称	描述
++$x	前递增	$x 加一递增，然后返回 $x
$x++	后递增	返回 $x，然后 $x 加一递增
--$x	前递减	$x 减一递减，然后返回 $x
$x--	后递减	返回 $x，然后 $x 减一递减

下例展示了使用不同递增/递减运算符的不同结果：

实例

```
<?php
$x=10;
echo ++$x; // 输出 11

$y=10;
echo $y++; // 输出 10

$z=5;
echo --$z; // 输出 4

$i=5;
echo $i--; // 输出 5
?>
```

PHP 比较运算符

PHP 比较运算符用于比较两个值（数字或字符串）：

运算符	名称	例子	结果
==	等于	$x == $y	如果 $x 等于 $y，则返回 true。
===	全等（完全相同）	$x === $y	如果 $x 等于 $y，且它们类型相同，则返回 true。
!=	不等于	$x != $y	如果 $x 不等于 $y，则返回 true。
<>	不等于	$x <> $y	如果 $x 不等于 $y，则返回 true。
!==	不全等（完全不同）	$x !== $y	如果 $x 不等于 $y，且它们类型不相同，则返回 true。
>	大于	$x > $y	如果 $x 大于 $y，则返回 true。
<	大于	$x < $y	如果 $x 小于 $y，则返回 true。
>=	大于或等于	$x >= $y	如果 $x 大于或者等于 $y，则返回 true.
<=	小于或等于	$x <= $y	如果 $x 小于或者等于 $y，则返回 true。

下例展示了使用某些比较运算符的不同结果：

实例

```
<?php
$x=100;
$y="100";

var_dump($x == $y);
echo "<br>";
var_dump($x === $y);
echo "<br>";
var_dump($x != $y);
echo "<br>";
var_dump($x !== $y);
echo "<br>";

$a=50;
$b=90;

var_dump($a > $b);
echo "<br>";
var_dump($a < $b);
?>
```

PHP 逻辑运算符

运算符	名称	例子	结果
and	与	$x and $y	如果 $x 和 $y 都为 true，则返回 true。
or	或	$x or $y	如果 $x 和 $y 至少有一个为 true，则返回 true。
xor	异或	$x xor $y	如果 $x 和 $y 有且仅有一个为 true，则返回 true。
&&	与	$x && $y	如果 $x 和 $y 都为 true，则返回 true。
||	或	$x || $y	如果 $x 和 $y 至少有一个为 true，则返回 true。
!	非	!$x	如果 $x 不为 true，则返回 true。

PHP 数组运算符

PHP 数组运算符用于比较数组：

运算符	名称	例子	结果
+	联合	$x + $y	$x 和 $y 的联合（但不覆盖重复的键）
==	相等	$x == $y	如果 $x 和 $y 拥有相同的键/值对，则返回 true。
===	全等	$x === $y	如果 $x 和 $y 拥有相同的键/值对，且顺序相同类型相同，则返回 true。
!=	不相等	$x != $y	如果 $x 不等于 $y，则返回 true。
<>	不相等	$x <> $y	如果 $x 不等于 $y，则返回 true。
!==	不全等	$x !== $y	如果 $x 与 $y 完全不同，则返回 true。

下例展示了使用不同数组运算符的不同结果：

实例

```
<?php
$x = array("a" => "red", "b" => "green");
$y = array("c" => "blue", "d" => "yellow");
$z = $x + $y; // $x 与 $y 的联合
var_dump($z);
var_dump($x == $y);
var_dump($x === $y);
var_dump($x != $y);
var_dump($x <> $y);
var_dump($x !== $y);
?>
```

PHP if...else...elseif 语句

条件语句用于基于不同条件执行不同的动作

PHP 条件语句

在您编写代码时，经常会希望为不同的决定执行不同的动作。您可以在代码中使用条件语句来实现这一点。

在 PHP 中，我们可以使用以下条件语句：

- *if* 语句 - 如果指定条件为真，则执行代码
- *if...else* 语句 - 如果条件为 true，则执行代码；如果条件为 false，则执行另一端代码
- *if...elseif....else* 语句 - 选择若干段代码块之一来执行
- *switch* 语句 - 语句多个代码块之一来执行

PHP - if 语句

if 语句用于在指定条件为 *true* 时执行代码。

语法

```
if (条件) {
  当条件为 true 时执行的代码;
}
```

下例将输出 "Have a good day!"，如果当前时间 (HOUR) 小于 20：

实例

```
<?php
$t=date("H");

if ($t<"20") {
  echo "Have a good day!";
}
?>
```

PHP - if...else 语句

请使用 if....else 语句在条件为 *true* 时执行代码，在条件为 *false* 时执行另一段代码。

语法

```
if (条件) {
  条件为 true 时执行的代码;
} else {
  条件为 false 时执行的代码;
}
```

下例将输出 "Have a good day!"，如果当前时间 (HOUR) 小于 20，否则输出 "Have a good night!"：

实例

```
<?php
$t=date("H");

if ($t<"20") {
  echo "Have a good day!";
} else {
  echo "Have a good night!";
}
?>
```

PHP - if...elseif....else 语句

请使用 if....elseif...else 语句来选择若干代码块之一来执行。

语法

```
if (条件) {
  条件为 true 时执行的代码;
} elseif (condition) {
  条件为 true 时执行的代码;
} else {
  条件为 false 时执行的代码;
}
```

下例将输出 "Have a good morning!"，如果当前时间 (HOUR) 小于 10，如果当前时间小于 20，则输出 "Have a good day!"。否则将输出 "Have a good night!"：

实例

```
<?php
$t=date("H");

if ($t<"10") {
  echo "Have a good morning!";
} elseif ($t<"20") {
  echo "Have a good day!";
} else {
  echo "Have a good night!";
}
?>
```

PHP - switch 语句

我们在下一节中学习 switch 语句。

PHP Switch 语句

switch 语句用于基于不同条件执行不同动作。

Switch 语句

如果您希望有选择地执行若干代码块之一，请使用 Switch 语句。

使用 Switch 语句可以避免冗长的 if..elseif..else 代码块。

语法

```
switch (expression)
{
case label1:
  code to be executed if expression = label1;
  break;
case label2:
  code to be executed if expression = label2;
  break;
default:
  code to be executed
  if expression is different
  from both label1 and label2;
}
```

工作原理：

1. 对表达式（通常是变量）进行一次计算
2. 把表达式的值与结构中 case 的值进行比较
3. 如果存在匹配，则执行与 case 关联的代码
4. 代码执行后，*break* 语句阻止代码跳入下一个 case 中继续执行
5. 如果没有 case 为真，则使用 default 语句

实例

```
<?php
switch ($x)
{
case 1:
  echo "Number 1";
  break;
case 2:
  echo "Number 2";
  break;
case 3:
  echo "Number 3";
  break;
default:
  echo "No number between 1 and 3";
}
?>

</body>
</html>
```

PHP while 循环

PHP while 循环在指定条件为 **true** 时执行代码块。

PHP 循环

在您编写代码时，经常需要反复运行同一代码块。我们可以使用循环来执行这样的任务，而不是在脚本中添加若干几乎相等的代码行。

在 PHP 中，我们有以下循环语句：

- *while* - 只要指定条件为真，则循环代码块
- *do...while* - 先执行一次代码块，然后只要指定条件为真则重复循环
- *for* - 循环代码块指定次数
- *foreach* - 遍历数组中的每个元素并循环代码块

PHP while 循环

只要指定的条件为真，while 循环就会执行代码块。

语法

```
while (条件为真) {
  要执行的代码;
}
```

上例首先把变量 $x 设置为 1（$x=1）。然后执行 while 循环，只要 $x 小于或等于 5。循环每运行一次，$x 将递增 1：

实例

```
<?php
$x=1;

while($x<=5) {
  echo "这个数字是：$x <br>";
  $x++;
}
?>
```

PHP do...while 循环

do...while 循环首先会执行一次代码块，然后检查条件，如果指定条件为真，则重复循环。

语法

```
do {
  要执行的代码;
} while (条件为真);
```

下面的例子首先把变量 $x 设置为 1（$x=1）。然后，do while 循环输出一段字符串，然后对变量 $x 递增 1。随后对条件进行检查（$x 是否小于或等于 5）。只要 $x 小于或等于 5，循环将会继续运行：

实例

```
<?php
$x=1;

do {
  echo "这个数字是：$x <br>";
  $x++;
} while ($x<=5);
?>
```

请注意，do while 循环只在执行循环内的语句之后才对条件进行测试。这意味着 do while 循环至少会执行一次语句，即使条件测试在第一次就失败了。

下面的例子把 $x 设置为 6，然后运行循环，随后对条件进行检查：

实例

```
<?php
$x=6;

do {
  echo "这个数字是：$x <br>";
  $x++;
} while ($x<=5);
?>
```

下一节会讲解 for 循环和 foreach 循环。

PHP for 循环

PHP for 循环执行代码块指定的次数。

PHP for 循环

如果您已经提前确定脚本运行的次数，可以使用 for 循环。

语法

```
for (init counter; test counter; increment counter) {
  code to be executed;
}
```

参数：

- init counter：初始化循环计数器的值
- test counter：: 评估每个循环迭代。如果值为 TRUE，继续循环。如果它的值为 FALSE，循环结束。
- increment counter：增加循环计数器的值

下面的例子显示了从 0 到 10 的数字：

实例

```
<?php
for ($x=0; $x<=10; $x++) {
  echo "数字是：$x <br>";
}
?>
```

PHP foreach 循环

foreach 循环只适用于数组，并用于遍历数组中的每个键/值对。

语法

```
foreach ($array as $value) {
  code to be executed;
}
```

每进行一次循环迭代，当前数组元素的值就会被赋值给 $value 变量，并且数组指针会逐一地移动，直到到达最后一个数组元素。

下面的例子演示的循环将输出给定数组（$colors）的值：

实例

```
<?php
$colors = array("red","green","blue","yellow");

foreach ($colors as $value) {
  echo "$value <br>";
}
?>
```

在稍后的章节，您将学到更多有关数组的知识。

PHP 函数

PHP 的真正力量来自它的函数：它拥有超过 **1000** 个内建的函数。

PHP 用户定义函数

除了内建的 PHP 函数，我们可以创建我们自己的函数。

函数是可以在程序中重复使用的语句块。

页面加载时函数不会立即执行。

函数只有在被调用时才会执行。

在 PHP 创建用户定义函数

用户定义的函数声明以关单 "function" 开头：

语法

```
function functionName() {
  被执行的代码;
}
```

注释：函数名能够以字母或下划线开头（而非数字）。

注释：函数名对大小写不敏感。

提示：函数名应该能够反映函数所执行的任务。

在下面的例子中，我们创建名为 "writeMsg()" 的函数。打开的花括号（{）指示函数代码的开始，而关闭的花括号（}）指示函数的结束。此函数输出 "Hello world!"。如需调用该函数，只要使用函数名即可：

实例

```
<?php
function writeMsg() {
  echo "Hello world!";
}

writeMsg(); // 调用函数
?>
```

PHP 函数参数

可以通过参数向函数传递信息。参数类似变量。

参数被定义在函数名之后，括号内部。您可以添加任意多参数，只要用逗号隔开即可。

下面的例子中的函数有一个参数（$fname）。当调用 familyName() 函数时，我们同时要传递一个名字（例如 Bill），这样会输出不同的名字，但是姓氏相同：

实例

```
<?php
function familyName($fname) {
  echo "$fname Zhang.<br>";
}

familyName("Li");
familyName("Hong");
familyName("Tao");
familyName("Xiao Mei");
familyName("Jian");
?>
```

下面的例子中的函数有两个参数（$fname 和 $year）：

实例

```
<?php
function familyName($fname,$year) {
  echo "$fname Zhang. Born in $year <br>";
}

familyName("Li","1975");
familyName("Hong","1978");
familyName("Tao","1983");
?>
```

PHP 默认参数值

下面的例子展示了如何使用默认参数。如果我们调用没有参数的 setHeight() 函数，它的参数会取默认值：

实例

```
<?php
function setHeight($minheight=50) {
  echo "The height is : $minheight <br>";
}

setHeight(350);
setHeight(); // 将使用默认值 50
setHeight(135);
setHeight(80);
?>
```

PHP 函数 - 返回值

如需使函数返回值，请使用 return 语句：

实例

```
<?php
function sum($x,$y) {
  $z=$x+$y;
  return $z;
}

echo "5 + 10 = " . sum(5,10) . "<br>";
echo "7 + 13 = " . sum(7,13) . "<br>";
echo "2 + 4 = " . sum(2,4);
?>
```

PHP 数组

数组能够在单独的变量名中存储一个或多个值。

实例

数组在单个变量中存储多个值：

```
<?php
$cars=array("Volvo","BMW","SAAB");
echo "I like " . $cars[0] . ", " . $cars[1] . " and " . $cars[2] . ".";
?>
```

什么是数组？

数组是特殊的变量，它可以同时保存一个以上的值。

如果您有一个项目列表（例如汽车品牌列表），在单个变量中存储这些品牌名称是这样的：

```
$cars1="Volvo";
$cars2="BMW";
$cars3="SAAB";
```

不过，假如您希望对变量进行遍历并找出特定的那个值？或者如果您需要存储 300 个汽车品牌，而不是 3 个呢？

解决方法是创建数组！

数组能够在单一变量名中存储许多值，并且您能够通过引用下标号来访问某个值。

在 PHP 中创建数组

在 PHP 中， array() 函数用于创建数组：

```
array();
```

在 PHP 中，有三种数组类型：

- 索引数组 - 带有数字索引的数组
- 关联数组 - 带有指定键的数组
- 多维数组 - 包含一个或多个数组的数组

PHP 索引数组

有两种创建索引数组的方法：

索引是自动分配的（索引从 0 开始）：

```
$cars=array("Volvo","BMW","SAAB");
```

或者也可以手动分配索引：

```
$cars[0]="Volvo";
$cars[1]="BMW";
$cars[2]="SAAB";
```

下面的例子创建名为 $cars 的索引数组，为其分配三个元素，然后输出包含数组值的一段文本：

实例

```
<?php
$cars=array("Volvo","BMW","SAAB");
echo "I like " . $cars[0] . ", " . $cars[1] . " and " . $cars[2] . ".";
?>
```

获得数组的长度 - count() 函数

count() 函数用于返回数组的长度（元素数）：

实例

```
<?php
$cars=array("Volvo","BMW","SAAB");
echo count($cars);
?>
```

遍历索引数组

如需遍历并输出索引数组的所有值，您可以使用 for 循环，就像这样：

实例

```
<?php
$cars=array("Volvo","BMW","SAAB");
$arrlength=count($cars);

for($x=0;$x<$arrlength;$x++) {
  echo $cars[$x];
  echo "<br>";
}
?>
```

PHP 关联数组

关联数组是使用您分配给数组的指定键的数组。

有两种创建关联数组的方法：

```
$age=array("Peter"=>"35","Ben"=>"37","Joe"=>"43");
```

或者：

```
$age['Peter']="35";
$age['Ben']="37";
$age['Joe']="43";
```

随后可以在脚本中使用指定键：

实例

```
<?php
$age=array("Bill"=>"35","Steve"=>"37","Peter"=>"43");
echo "Peter is " . $age['Peter'] . " years old.";
?>
```

遍历关联数组

如需遍历并输出关联数组的所有值，您可以使用 foreach 循环，就像这样：

实例

```
<?php
$age=array("Bill"=>"35","Steve"=>"37","Peter"=>"43");

foreach($age as $x=>$x_value) {
  echo "Key=" . $x . ", Value=" . $x_value;
  echo "<br>";
}
?>
```

多维数组

我们将在 PHP 高级教程中讲解多维数组。

完整的 PHP 数组参考手册

如需完整的数组函数参考手册，请访问我们的 PHP 数组参考手册。

该参考手册包含每个函数的简要描述、使用示例。

PHP 数组排序

数组中的元素能够以字母或数字顺序进行升序或降序排序。

PHP - 数组的排序函数

在本节中，我们将学习如下 PHP 数组排序函数：

- sort() - 以升序对数组排序
- rsort() - 以降序对数组排序
- asort() - 根据值，以升序对关联数组进行排序
- ksort() - 根据键，以升序对关联数组进行排序
- arsort() - 根据值，以降序对关联数组进行排序
- krsort() - 根据键，以降序对关联数组进行排序

对数组进行升序排序 - sort()

下面的例子按照字母升序对数组 $cars 中的元素进行排序：

实例

```
<?php
$cars=array("Volvo","BMW","SAAB");
sort($cars);
?>
```

下面的例子按照数字升序对数组 $numbers 中的元素进行排序：

实例

```
<?php
$numbers=array(3,5,1,22,11);
sort($numbers);
?>
```

对数组进行降序排序 - rsort()

下面的例子按照字母降序对数组 $cars 中的元素进行排序：

实例

```
<?php
$cars=array("Volvo","BMW","SAAB");
rsort($cars);
?>
```

下面的例子按照数字降序对数组 $numbers 中的元素进行排序：

实例

```
<?php
$numbers=array(3,5,1,22,11);
rsort($numbers);
?>
```

根据值对数组进行升序排序 - asort()

下面的例子根据值对关联数组进行升序排序：

实例

```
<?php
$age=array("Bill"=>"35","Steve"=>"37","Peter"=>"43");
asort($age);
?>
```

根据键对数组进行升序排序 - ksort()

下面的例子根据键对关联数组进行升序排序：

实例

```
<?php
$age=array("Bill"=>"35","Steve"=>"37","Peter"=>"43");
ksort($age);
?>
```

根据值对数组进行降序排序 - arsort()

下面的例子根据值对关联数组进行降序排序：

实例

```
<?php
$age=array("Bill"=>"35","Steve"=>"37","Peter"=>"43");
arsort($age);
?>
```

根据键对数组进行降序排序 - krsort()

下面的例子根据键对关联数组进行降序排序：

实例

```
<?php
$age=array("Bill"=>"35","Steve"=>"37","Peter"=>"43");
krsort($age);
?>
```

完整的 PHP 数组参考手册

如需完整的数组函数参考手册，请访问我们的 PHP 数组参考手册。

该参考手册包含每个函数的简要描述、使用示例。

PHP 全局变量 - 超全局变量

超全局变量 在 **PHP 4.1.0** 中引入，是在全部作用域中始终可用的内置变量。

PHP 全局变量 - 超全局变量

PHP 中的许多预定义变量都是“超全局的”，这意味着它们在一个脚本的全部作用域中都可用。在函数或方法中无需执行 global $variable; 就可以访问它们。

这些超全局变量是：

- $GLOBALS
- $_SERVER
- $_REQUEST
- $_POST
- $_GET
- $_FILES
- $_ENV
- $_COOKIE
- $_SESSION

本节会介绍一些超全局变量，并会在稍后的章节讲解其他的超全局变量。

$GLOBALS — 引用全局作用域中可用的全部变量

$GLOBALS 这种全局变量用于在 PHP 脚本中的任意位置访问全局变量（从函数或方法中均可）。

PHP 在名为 $GLOBALS[index] 的数组中存储了所有全局变量。变量的名字就是数组的键。

下面的例子展示了如何使用超级全局变量 $GLOBALS：

实例

```
<?php
$x = 75;
$y = 25;

function addition() {
  $GLOBALS['z'] = $GLOBALS['x'] + $GLOBALS['y'];
}

addition();
echo $z;
?>
```

在上面的例子中，由于 z 是 $GLOBALS 数组中的变量，因此在函数之外也可以访问它。

PHP $_SERVER

$_SERVER 这种超全局变量保存关于报头、路径和脚本位置的信息。

下面的例子展示了如何使用 $_SERVER 中的某些元素：

实例

```
<?php
echo $_SERVER['PHP_SELF'];
echo "<br>";
echo $_SERVER['SERVER_NAME'];
echo "<br>";
echo $_SERVER['HTTP_HOST'];
echo "<br>";
echo $_SERVER['HTTP_REFERER'];
echo "<br>";
echo $_SERVER['HTTP_USER_AGENT'];
echo "<br>";
echo $_SERVER['SCRIPT_NAME'];
?>
```

下表列出了您能够在 $_SERVER 中访问的最重要的元素：

元素/代码	描述
$_SERVER['PHP_SELF']	返回当前执行脚本的文件名。
$_SERVER['GATEWAY_INTERFACE']	返回服务器使用的 CGI 规范的版本。
$_SERVER['SERVER_ADDR']	返回当前运行脚本所在的服务器的 IP 地址。
$_SERVER['SERVER_NAME']	返回当前运行脚本所在的服务器的主机名（比如 www.w3school.com.cn）。
$_SERVER['SERVER_SOFTWARE']	返回服务器标识字符串（比如 Apache/2.2.24）。
$_SERVER['SERVER_PROTOCOL']	返回请求页面时通信协议的名称和版本（例如，“HTTP/1.0”）。

$_SERVER['REQUEST_METHOD']	返回访问页面使用的请求方法（例如 POST）。
$_SERVER['REQUEST_TIME']	返回请求开始时的时间戳（例如 1577687494）。
$_SERVER['QUERY_STRING']	返回查询字符串，如果是通过查询字符串访问此页面。
$_SERVER['HTTP_ACCEPT']	返回来自当前请求的请求头。
$_SERVER['HTTP_ACCEPT_CHARSET']	返回来自当前请求的 Accept_Charset 头（例如 utf-8,ISO-8859-1）
$_SERVER['HTTP_HOST']	返回来自当前请求的 Host 头。
$_SERVER['HTTP_REFERER']	返回当前页面的完整 URL（不可靠，因为不是所有用户代理都支持）。
$_SERVER['HTTPS']	是否通过安全 HTTP 协议查询脚本。
$_SERVER['REMOTE_ADDR']	返回浏览当前页面的用户的 IP 地址。
$_SERVER['REMOTE_HOST']	返回浏览当前页面的用户的主机名。
$_SERVER['REMOTE_PORT']	返回用户机器上连接到 Web 服务器所使用的端口号。
$_SERVER['SCRIPT_FILENAME']	返回当前执行脚本的绝对路径。
$_SERVER['SERVER_ADMIN']	该值指明了 Apache 服务器配置文件中的 SERVER_ADMIN 参数。
$_SERVER['SERVER_PORT']	Web 服务器使用的端口。默认值为 “80”。
$_SERVER['SERVER_SIGNATURE']	返回服务器版本和虚拟主机名。
$_SERVER['PATH_TRANSLATED']	当前脚本所在文件系统（非文档根目录）的基本路径。
$_SERVER['SCRIPT_NAME']	返回当前脚本的路径。
$_SERVER['SCRIPT_URI']	返回当前页面的 URI。

PHP $_REQUEST

PHP $_REQUEST 用于收集 HTML 表单提交的数据。

下面的例子展示了一个包含输入字段及提交按钮的表单。当用户通过点击提交按钮来提交表单数据时, 表单数据将发送到 <form> 标签的 action 属性中指定的脚本文件。在这个例子中，我们指定文件本身来处理表单数据。如果您需要使用其他的 PHP 文件来处理表单数据，请修改为您选择的文件名即可。然后，我们可以使用超级全局变量 $_REQUEST 来收集 input 字段的值：

实例

```
<html>
<body>

<form method="post" action="<?php echo $_SERVER['PHP_SELF'];?>">
Name: <input type="text" name="fname">
<input type="submit">
</form>

<?php
$name = $_REQUEST['fname'];
echo $name;
?>

</body>
</html>
```

PHP $_POST

PHP $_POST 广泛用于收集提交 method="post" 的 HTML 表单后的表单数据。$_POST 也常用于传递变量。

下面的例子展示了一个包含输入字段和提交按钮的表单。当用户点击提交按钮来提交数据后，表单数据会发送到 <form> 标签的 action 属性中指定的文件。在本例中，我们指定文件本身来处理表单数据。如果您希望使用另一个 PHP 页面来处理表单数据，请用更改为您选择的文件名。然后，我们可以使用超全局变量 $_POST 来收集输入字段的值：

实例

```
<html>
<body>

<form method="post" action="<?php echo $_SERVER['PHP_SELF'];?>">
Name: <input type="text" name="fname">
<input type="submit">
</form>

<?php
$name = $_POST['fname'];
echo $name;
?>

</body>
</html>
```

PHP $_GET

PHP $_GET 也可用于收集提交 HTML 表单 (method="get") 之后的表单数据。

$_GET 也可以收集 URL 中的发送的数据。

假设我们有一张页面含有带参数的超链接：

```
<html>
<body>

<a href="test_get.php?subject=PHP&web=W3school.com.cn">测试 $GET</a>

</body>
</html>
```

当用户点击链接 "Test $GET"，参数 "subject" 和 "web" 被发送到 "test_get.php"，然后您就能够通过 $_GET 在 "test_get.php" 中访问这些值了。

下面的例子是 "test_get.php" 中的代码：

实例

```
<html>
<body>

<?php
echo "Study " . $_GET['subject'] . " at " . $_GET['web'];
?>

</body>
</html>
```

提示：您将在 PHP 表单这一节中学到更多有关 $_POST 和 $_GET 的知识。

PHP 魔术变量

PHP 向它运行的任何脚本提供了大量的预定义常量。

不过很多常量都是由不同的扩展库定义的，只有在加载了这些扩展库时才会出现，或者动态加载后，或者在编译时已经包括进去了。

有八个魔术常量它们的值随着它们在代码中的位置改变而改变。

例如 __LINE__ 的值就依赖于它在脚本中所处的行来决定。这些特殊的常量不区分大小写，如下：

__LINE__

文件中的当前行号。

实例:

```
<?php
echo '这是第 “ '  . __LINE__ . ' ” 行';
?>
```

以上实例输出结果为：

```
这是第 “ 2 ” 行
```

__FILE__

文件的完整路径和文件名。如果用在被包含文件中，则返回被包含的文件名。

自 PHP 4.0.2 起，**FILE** 总是包含一个绝对路径（如果是符号连接，则是解析后的绝对路径），而在此之前的版本有时会包含一个相对路径。

实例:

```
<?php
echo '该文件位于 “ '  . __FILE__ . ' ” ';
?>
```

以上实例输出结果为：

```
该文件位于 “ E:\wamp\www\test\index.php ”
```

__DIR__

文件所在的目录。如果用在被包括文件中，则返回被包括的文件所在的目录。

它等价于 dirname(**FILE**)。除非是根目录，否则目录中名不包括末尾的斜杠。（PHP 5.3.0中新增）

实例:

```
<?php
echo '该文件位于 " '  . __DIR__ . ' " ';
?>
```

以上实例输出结果为：

```
该文件位于 " E:\wamp\www\test "
```

__FUNCTION__

函数名称（PHP 4.3.0 新加）。自 PHP 5 起本常量返回该函数被定义时的名字（区分大小写）。在 PHP 4 中该值总是小写字母的。

实例:

```
<?php
function test() {
    echo  '函数名为：' . __FUNCTION__ ;
}
test();
?>
```

以上实例输出结果为：

```
函数名为：test
```

__CLASS__

类的名称（PHP 4.3.0 新加）。自 PHP 5 起本常量返回该类被定义时的名字（区分大小写）。

在 PHP 4 中该值总是小写字母的。类名包括其被声明的作用区域（例如 Foo\Bar）。注意自 PHP 5.4 起 **CLASS** 对 trait 也起作用。当用在 trait 方法中时，**CLASS** 是调用 trait 方法的类的名字。

实例:

```
<?php
<?php
class test {
    function _print() {
        echo '类名为：'  . __CLASS__ . "<br>";
        echo  '函数名为：' . __FUNCTION__ ;
    }
}
$t = new test();
$t->_print();
?>
```

以上实例输出结果为：

```
类名为：test
函数名为：_print
```

__TRAIT__

Trait 的名字（PHP 5.4.0 新加）。自 PHP 5.4.0 起，PHP 实现了代码复用的一个方法，称为 traits。

Trait 名包括其被声明的作用区域（例如 Foo\Bar）。

从基类继承的成员被插入的 SayWorld Trait 中的 MyHelloWorld 方法所覆盖。其行为 MyHelloWorld 类中定义的方法一致。优先顺序是当前类中的方法会覆盖 trait 方法，而 trait 方法又覆盖了基类中的方法。

```
<?php
class Base {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait SayWorld {
    public function sayHello() {
        parent::sayHello();
        echo 'World!';
    }
}

class MyHelloWorld extends Base {
    use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
?>
```

以上例程会输出：

```
Hello World!
```

__METHOD__

类的方法名（PHP 5.0.0 新加）。返回该方法被定义时的名字（区分大小写）。

实例:

```
<?php
function test() {
    echo  '函数名为：' . __METHOD__ ;
}
test();
?>
```

以上实例输出结果为：

```
函数名为：test
```

__NAMESPACE__

当前命名空间的名称（区分大小写）。此常量是在编译时定义的（PHP 5.3.0 新增）。

实例:

```
<?php
namespace MyProject;

echo '命名空间为："', __NAMESPACE__, '"'; // 输出 "MyProject"
?>
```

以上实例输出结果为：

```
命名空间为："MyProject"
```

PHP 命名空间(namespace)

PHP 命名空间(namespace)是在PHP 5.3中加入的，如果你学过C#和Java，那命名空间就不算什么新事物。 不过在PHP当中还是有着相当重要的意义。

PHP 命名空间可以解决以下两类问题：

1. 用户编写的代码与PHP内部的类/函数/常量或第三方类/函数/常量之间的名字冲突。
2. 为很长的标识符名称(通常是为了缓解第一类问题而定义的)创建一个别名（或简短）的名称，提高源代码的可读性。

定义命名空间

默认情况下，所有常量、类和函数名都放在全局空间下，就和PHP支持命名空间之前一样。

命名空间通过关键字namespace 来声明。如果一个文件中包含命名空间，它必须在其它所有代码之前声明命名空间。语法格式如下；

```
< ?php
// 定义代码在 'MyProject' 命名空间中
namespace MyProject;

// ... 代码 ...
```

你也可以在同一个文件中定义不同的命名空间代码，如：

```
< ?php
namespace MyProject1;
// MyProject1 命名空间中的PHP代码

namespace MyProject2;
// MyProject2 命名空间中的PHP代码

// 另一种语法
namespace MyProject3 {
 // MyProject3 命名空间中的PHP代码
}
?>
```

在声明命名空间之前唯一合法的代码是用于定义源文件编码方式的 declare 语句。所有非PHP 代码包括空白符都不能出现在命名空间的声明之前。

```
<?php
declare(encoding='UTF-8'); //定义多个命名空间和不包含在命名空间中的代码
namespace MyProject {

const CONNECT_OK = 1;
class Connection { /* ... */ }
function connect() { /* ... */  }
}

namespace { // 全局代码
session_start();
$a = MyProject\connect();
echo MyProject\Connection::start();
}
?>
```

以下代码会出现语法错误：

```
<html>
<?php
namespace MyProject; // 命名空间前出现了“<html>” 会致命错误 - 命名空间必须是程序脚本的第一条语句
?>
```

子命名空间

与目录和文件的关系很象，PHP 命名空间也允许指定层次化的命名空间的名称。因此，命名空间的名字可以使用分层次的方式定义：

```
<?php
namespace MyProject\Sub\Level;  //声明分层次的单个命名空间

const CONNECT_OK = 1;
class Connection { /* ... */ }
function Connect() { /* ... */  }

?>
```

上面的例子创建了常量 MyProject\Sub\Level\CONNECT_OK，类 MyProject\Sub\Level\Connection 和函数 MyProject\Sub\Level\Connection。

命名空间使用

PHP 命名空间中的类名可以通过三种方式引用：

1. 非限定名称，或不包含前缀的类名称，例如 $a=new foo(); 或 foo::staticmethod();。如果当前命名空间是 currentnamespace，foo 将被解析为 currentnamespace\foo。如果使用 foo 的代码是全局的，不包含在任何命名空间中的代码，则 foo 会被解析为foo。 警告：如果命名空间中的函数或常量未定义，则该非限定的函数名称或常量名称会被解析为全局函数名称或常量名称。

2. 限定名称,或包含前缀的名称，例如 $a = new subnamespace\foo(); 或 subnamespace\foo::staticmethod();。如果当前的命名空间是 currentnamespace，则 foo 会被解析为 currentnamespace\subnamespace\foo。如果使用 foo 的代码是全局的，不包含在任何命名空间中的代码，foo 会被解析为subnamespace\foo。 完全限定名称，或包含了全局前缀操作符的名称，例如， $a = new \currentnamespace\foo(); 或 \currentnamespace\foo::staticmethod();。在这种情况下，foo 总是被解析为代码中的文字名(literal name)currentnamespace\foo。

下面是一个使用这三种方式的实例：

file1.php 文件代码

```
<?php
namespace Foo\Bar\subnamespace;

const FOO = 1;
function foo() {}
class foo
{
    static function staticmethod() {}
}
?>
```

file2.php 文件代码

```
<?php
namespace Foo\Bar;
include 'file1.php';

const FOO = 2;
function foo() {}
class foo
{
    static function staticmethod() {}
}

/* 非限定名称 */
foo(); // 解析为 Foo\Bar\foo resolves to function Foo\Bar\foo
foo::staticmethod(); // 解析为类 Foo\Bar\foo的静态方法staticmethod。resolves to class Foo\Ba
echo FOO; // resolves to constant Foo\Bar\FOO

/* 限定名称 */
subnamespace\foo(); // 解析为函数 Foo\Bar\subnamespace\foo
subnamespace\foo::staticmethod(); // 解析为类 Foo\Bar\subnamespace\foo,
                                  // 以及类的方法 staticmethod
echo subnamespace\FOO; // 解析为常量 Foo\Bar\subnamespace\FOO

/* 完全限定名称 */
\Foo\Bar\foo(); // 解析为函数 Foo\Bar\foo
\Foo\Bar\foo::staticmethod(); // 解析为类 Foo\Bar\foo, 以及类的方法 staticmethod
echo \Foo\Bar\FOO; // 解析为常量 Foo\Bar\FOO
?>
```

注意访问任意全局类、函数或常量，都可以使用完全限定名称，例如 \strlen() 或 \Exception 或 \INI_ALL。

在命名空间内部访问全局类、函数和常量：

```
<?php
namespace Foo;

function strlen() {}
const INI_ALL = 3;
class Exception {}

$a = \strlen('hi'); // 调用全局函数strlen
$b = \INI_ALL; // 访问全局常量 INI_ALL
$c = new \Exception('error'); // 实例化全局类 Exception
?>
```

命名空间和动态语言特征

PHP 命名空间的实现受到其语言自身的动态特征的影响。因此，如果要将下面的代码转换到命名空间中，动态访问元素。

example1.php 文件代码：

```
<?php
class classname
{
    function __construct()
    {
        echo __METHOD__,"\n";
    }
}
function funcname()
{
    echo __FUNCTION__,"\n";
}
const constname = "global";

$a = 'classname';
$obj = new $a; // prints classname::__construct
$b = 'funcname';
$b(); // prints funcname
echo constant('constname'), "\n"; // prints global
?>
```

必须使用完全限定名称（包括命名空间前缀的类名称）。注意因为在动态的类名称、函数名称或常量名称中，限定名称和完全限定名称没有区别，因此其前导的反斜杠是不必要的。

动态访问命名空间的元素

```
<?php
namespace namespacename;
class classname
{
    function __construct()
    {
        echo __METHOD__,"\n";
    }
}
function funcname()
{
    echo __FUNCTION__,"\n";
}
const constname = "namespaced";

include 'example1.php';

$a = 'classname';
$obj = new $a; // prints classname::__construct
$b = 'funcname';
$b(); // prints funcname
echo constant('constname'), "\n"; // prints global

/* note that if using double quotes, "\\namespacename\\classname" must be used */
$a = '\namespacename\classname';
$obj = new $a; // prints namespacename\classname::__construct
$a = 'namespacename\classname';
$obj = new $a; // also prints namespacename\classname::__construct
$b = 'namespacename\funcname';
$b(); // prints namespacename\funcname
$b = '\namespacename\funcname';
$b(); // also prints namespacename\funcname
echo constant('\namespacename\constname'), "\n"; // prints namespaced
echo constant('namespacename\constname'), "\n"; // also prints namespaced
?>
```

namespace关键字和NAMESPACE常量

PHP支持两种抽象的访问当前命名空间内部元素的方法，**NAMESPACE** 魔术常量和namespace关键字。

常量**NAMESPACE**的值是包含当前命名空间名称的字符串。在全局的，不包括在任何命名空间中的代码，它包含一个空的字符串。

NAMESPACE 示例, 在命名空间中的代码

```
<?php
namespace MyProject;

echo '"', __NAMESPACE__, '"'; // 输出 "MyProject"
?>
```

NAMESPACE 示例，全局代码

```
<?php

echo '"', __NAMESPACE__, '"'; // 输出 ""
?>
```

常量 **NAMESPACE** 在动态创建名称时很有用，例如：

使用**NAMESPACE**动态创建名称

```
<?php
namespace MyProject;

function get($classname)
{
    $a = __NAMESPACE__ . '\\' . $classname;
    return new $a;
}
?>
```

关键字 namespace 可用来显式访问当前命名空间或子命名空间中的元素。它等价于类中的 self 操作符。

amespace操作符，命名空间中的代码

```
<?php
namespace MyProject;

use blah\blah as mine; // see "Using namespaces: importing/aliasing"

blah\mine(); // calls function blah\blah\mine()
namespace\blah\mine(); // calls function MyProject\blah\mine()

namespace\func(); // calls function MyProject\func()
namespace\sub\func(); // calls function MyProject\sub\func()
namespace\cname::method(); // calls static method "method" of class MyProject\cname
$a = new namespace\sub\cname(); // instantiates object of class MyProject\sub\cname
$b = namespace\CONSTANT; // assigns value of constant MyProject\CONSTANT to $b
?>
```

namespace操作符, 全局代码

```
<?php

namespace\func(); // calls function func()
namespace\sub\func(); // calls function sub\func()
namespace\cname::method(); // calls static method "method" of class cname
$a = new namespace\sub\cname(); // instantiates object of class sub\cname
$b = namespace\CONSTANT; // assigns value of constant CONSTANT to $b
?>
```

使用命名空间：别名/导入

PHP 命名空间支持 有两种使用别名或导入方式：为类名称使用别名，或为命名空间名称使用别名。注意PHP不支持导入函数或常量。

在PHP中，别名是通过操作符 use 来实现的. 下面是一个使用所有可能的三种导入方式的例子：

1、使用use操作符导入/使用别名

```
<?php
namespace foo;
use My\Full\Classname as Another;

// 下面的例子与 use My\Full\NSname as NSname 相同
use My\Full\NSname;

// 导入一个全局类
use \ArrayObject;

$obj = new namespace\Another; // 实例化 foo\Another 对象
$obj = new Another; // 实例化 My\Full\Classname  对象
NSname\subns\func(); // 调用函数 My\Full\NSname\subns\func
$a = new ArrayObject(array(1)); // 实例化 ArrayObject 对象
// 如果不使用 "use \ArrayObject" ，则实例化一个 foo\ArrayObject 对象
?>
```

2、 一行中包含多个use语句

```
<?php
use My\Full\Classname as Another, My\Full\NSname;

$obj = new Another; // 实例化 My\Full\Classname 对象
NSname\subns\func(); // 调用函数 My\Full\NSname\subns\func
?>
```

导入操作是在编译执行的，但动态的类名称、函数名称或常量名称则不是。

3、导入和动态名称

```
<?php
use My\Full\Classname as Another, My\Full\NSname;

$obj = new Another; // 实例化一个 My\Full\Classname 对象
$a = 'Another';
$obj = new $a;      // 实际化一个 Another 对象
?>
```

另外，导入操作只影响非限定名称和限定名称。完全限定名称由于是确定的，故不受导入的影响。

4、导入和完全限定名称

```
<?php
use My\Full\Classname as Another, My\Full\NSname;

$obj = new Another; // instantiates object of class My\Full\Classname
$obj = new \Another; // instantiates object of class Another
$obj = new Another\thing; // instantiates object of class My\Full\Classname\thing
$obj = new \Another\thing; // instantiates object of class Another\thing
?>
```

使用命名空间：后备全局函数/常量

在一个命名空间中，当 PHP 遇到一个非限定的类、函数或常量名称时，它使用不同的优先策略来解析该名称。类名称总是解析到当前命名空间中的名称。因此在访问系统内部或不包含在命名空间中的类名称时，必须使用完全限定名称，例如：

1、在命名空间中访问全局类

```
<?php
namespace A\B\C;
class Exception extends \Exception {}

$a = new Exception('hi'); // $a 是类 A\B\C\Exception 的一个对象
$b = new \Exception('hi'); // $b 是类 Exception 的一个对象

$c = new ArrayObject; // 致命错误, 找不到 A\B\C\ArrayObject 类
?>
```

对于函数和常量来说，如果当前命名空间中不存在该函数或常量，PHP 会退而使用全局空间中的函数或常量。

2、 命名空间中后备的全局函数/常量

```
<?php
namespace A\B\C;

const E_ERROR = 45;
function strlen($str)
{
    return \strlen($str) - 1;
}

echo E_ERROR, "\n"; // 输出 "45"
echo INI_ALL, "\n"; // 输出 "7" - 使用全局常量 INI_ALL

echo strlen('hi'), "\n"; // 输出 "1"
if (is_array('hi')) { // 输出 "is not array"
    echo "is array\n";
} else {
    echo "is not array\n";
}
?>
```

全局空间

如果没有定义任何命名空间，所有的类与函数的定义都是在全局空间，与 PHP 引入命名空间概念前一样。在名称前加上前缀 \ 表示该名称是全局空间中的名称，即使该名称位于其它的命名空间中时也是如此。

使用全局空间说明

```
<?php
namespace A\B\C;

/* 这个函数是 A\B\C\fopen */
function fopen() {
     /* ... */
     $f = \fopen(...); // 调用全局的fopen函数
     return $f;
}
?>
```

命名空间的顺序

自从有了命名空间之后，最容易出错的该是使用类的时候，这个类的寻找路径是什么样的了。

```
<?php
namespace A;
use B\D, C\E as F;

// 函数调用

foo();      // 首先尝试调用定义在命名空间"A"中的函数foo()
            // 再尝试调用全局函数 "foo"

\foo();     // 调用全局空间函数 "foo"

my\foo();   // 调用定义在命名空间"A\my"中函数 "foo"

F();        // 首先尝试调用定义在命名空间"A"中的函数 "F"
            // 再尝试调用全局函数 "F"

// 类引用

new B();    // 创建命名空间 "A" 中定义的类 "B" 的一个对象
            // 如果未找到，则尝试自动装载类 "A\B"

new D();    // 使用导入规则，创建命名空间 "B" 中定义的类 "D" 的一个对象
            // 如果未找到，则尝试自动装载类 "B\D"

new F();    // 使用导入规则，创建命名空间 "C" 中定义的类 "E" 的一个对象
            // 如果未找到，则尝试自动装载类 "C\E"

new \B();   // 创建定义在全局空间中的类 "B" 的一个对象
            // 如果未发现，则尝试自动装载类 "B"

new \D();   // 创建定义在全局空间中的类 "D" 的一个对象
            // 如果未发现，则尝试自动装载类 "D"

new \F();   // 创建定义在全局空间中的类 "F" 的一个对象
            // 如果未发现，则尝试自动装载类 "F"

// 调用另一个命名空间中的静态方法或命名空间函数

B\foo();    // 调用命名空间 "A\B" 中函数 "foo"

B::foo();   // 调用命名空间 "A" 中定义的类 "B" 的 "foo" 方法
            // 如果未找到类 "A\B" ，则尝试自动装载类 "A\B"

D::foo();   // 使用导入规则，调用命名空间 "B" 中定义的类 "D" 的 "foo" 方法
            // 如果类 "B\D" 未找到，则尝试自动装载类 "B\D"

\B\foo();   // 调用命名空间 "B" 中的函数 "foo"

\B::foo();  // 调用全局空间中的类 "B" 的 "foo" 方法
            // 如果类 "B" 未找到，则尝试自动装载类 "B"

// 当前命名空间中的静态方法或函数

A\B::foo();   // 调用命名空间 "A\A" 中定义的类 "B" 的 "foo" 方法
              // 如果类 "A\A\B" 未找到，则尝试自动装载类 "A\A\B"

\A\B::foo();  // 调用命名空间 "A\B" 中定义的类 "B" 的 "foo" 方法
              // 如果类 "A\B" 未找到，则尝试自动装载类 "A\B"
?>
```

名称解析遵循下列规则：

1. 对完全限定名称的函数，类和常量的调用在编译时解析。例如 *new \A\B* 解析为类 *A\B*。
2. 所有的非限定名称和限定名称（非完全限定名称）根据当前的导入规则在编译时进行转换。例如，如果命名空间 *A\B\C* 被导入为 *C*，那么对 *C\D\e()* 的调用就会被转换为 *A\B\C\D\e()*。

3. 在命名空间内部，所有的没有根据导入规则转换的限定名称均会在其前面加上当前的命名空间名称。例如，在命名空间 *A\B* 内部调用 *C\D\e()*，则 *C\D\e()* 会被转换为 *A\B\C\D\e()* 。
4. 非限定类名根据当前的导入规则在编译时转换（用全名代替短的导入名称）。例如，如果命名空间 *A\B\C* 导入为C，则 *new C()* 被转换为 *new A\B\C()* 。
5. 在命名空间内部（例如A\B），对非限定名称的函数调用是在运行时解析的。例如对函数 *foo()* 的调用是这样解析的：
   i. 在当前命名空间中查找名为 *A\B\foo()* 的函数
   ii. 尝试查找并调用 全局*(global)* 空间中的函数 *foo()*。
6. 在命名空间（例如*A\B*）内部对非限定名称或限定名称类（非完全限定名称）的调用是在运行时解析的。下面是调用 *new C()* 及 *new D\E()* 的解析过程： *new C()*的解析:
   i. 在当前命名空间中查找*A\B\C*类。
   ii. 尝试自动装载类*A\B\C*。*new D\E()*的解析:
   iii. 在类名称前面加上当前命名空间名称变成：*A\B\D\E*，然后查找该类。
   iv. 尝试自动装载类 *A\B\D\E*。为了引用全局命名空间中的全局类，必须使用完全限定名称 *new \C()*。

PHP 表单

PHP 表单处理

PHP 超全局变量 **$_GET** 和 **$_POST** 用于收集表单数据（**form-data**）。

PHP - 一个简单的 HTML 表单

下面的例子显示了一个简单的 HTML 表单，它包含两个输入字段和一个提交按钮：

实例

```
<html>
<body>

<form action="welcome.php" method="post">
Name: <input type="text" name="name"><br>
E-mail: <input type="text" name="email"><br>
<input type="submit">
</form>

</body>
</html>
```

当用户填写此表单并点击提交按钮后，表单数据会发送到名为 "welcome.php" 的 PHP 文件供处理。表单数据是通过 HTTP POST 方法发送的。

如需显示出被提交的数据，您可以简单地输出（echo）所有变量。"welcome.php" 文件是这样的：

```
<html>
<body>

Welcome <?php echo $_POST["name"]; ?><br>
Your email address is: <?php echo $_POST["email"]; ?>

</body>
</html>
```

输出：

```
Welcome John
Your email address is john.doe@example.com
```

使用 HTTP GET 方法也能得到相同的结果：

实例

```
<html>
<body>

<form action="welcome_get.php" method="get">
Name: <input type="text" name="name"><br>
E-mail: <input type="text" name="email"><br>
<input type="submit">
</form>

</body>
</html>
```

"welcome_get.php" 是这样的：

```
<html>
<body>

Welcome <?php echo $_GET["name"]; ?><br>
Your email address is: <?php echo $_GET["email"]; ?>

</body>
</html>
```

上面的代码很简单。不过，最重要的内容被漏掉了。您需要对表单数据进行验证，以防止脚本出现漏洞。

注意：在处理 PHP 表单时请关注安全！

本页未包含任何表单验证程序，它只向我们展示如何发送并接收表单数据。

不过稍后的章节会为您讲解如何提高 PHP 表单的安全性！对表单适当的安全验证对于抵御黑客攻击和垃圾邮件非常重要！

GET vs. POST

GET 和 POST 都创建数组（例如，array(key => value, key2 => value2, key3 => value3, ...)）。此数组包含键/值对，其中的键是表单控件的名称，而值是来自用户的输入数据。

GET 和 POST 被视作 $_GET 和 $_POST。它们是超全局变量，这意味着对它们的访问无需考虑作用域 - 无需任何特殊代码，您能够从任何函数、类或文件访问它们。

$_GET 是通过 URL 参数传递到当前脚本的变量数组。

$_POST 是通过 HTTP POST 传递到当前脚本的变量数组。

何时使用 GET？

通过 GET 方法从表单发送的信息对任何人都是可见的（所有变量名和值都显示在 URL 中）。GET 对所发送信息的数量也有限制。限制在大于 2000 个字符。不过，由于变量显示在 URL 中，把页面添加到书签中也更为方便。

GET 可用于发送非敏感的数据。

注释：绝不能使用 GET 来发送密码或其他敏感信息！

何时使用 **POST**？

通过 POST 方法从表单发送的信息对其他人是不可见的（所有名称/值会被嵌入 HTTP 请求的主体中），并且对所发送信息的数量也无限制。

此外 POST 支持高阶功能，比如在向服务器上传文件时进行 multi-part 二进制输入。

不过，由于变量未显示在 URL 中，也就无法将页面添加到书签。

提示：开发者偏爱 POST 来发送表单数据。

接下来让我们看看如何安全地处理 PHP 表单！

PHP 表单验证

本节和下一节讲解如何使用 **PHP** 来验证表单数据。

PHP 表单验证

提示：在处理 PHP 表单时请重视安全性！

这些页面将展示如何安全地处理 PHP 表单。对 HTML 表单数据进行适当的验证对于防范黑客和垃圾邮件很重要！

我们稍后使用的 HTML 表单包含多种输入字段：必需和可选的文本字段、单选按钮以及提交按钮：

```
<h2>PHP 表单验证实例</h2>
<p><span class="error">* 必填字段</span></p>

<form method="post" action="/demo/demo_form_validation_complete.php">

姓名：
<input type="text" name="name" value="">
<span class="error">* </span>
<br><br>
电邮：
<input type="text" name="email" value="">
<span class="error">* </span>
<br><br>
网址：
<input type="text" name="website" value="">
<span class="error"></span>
<br><br>
<label>
评论：
<textarea name="comment" rows="5" cols="40"></textarea>
<br><br>
性别：
<input type="radio" name="gender"  value="female">女性
<input type="radio" name="gender"  value="male">男性
<span class="error">* </span>
<br><br>
<input type="submit" name="submit" value="提交">

</form>

<h2>您的输入：</h2>
```

上面的表单使用如下验证规则：

字段	验证规则
Name	必需。必须包含字母和空格。
E-mail	必需。必须包含有效的电子邮件地址（包含 @ 和 .）。
Website	可选。如果选填，则必须包含有效的 URL。
Comment	可选。多行输入字段（文本框）。
Gender	必需。必须选择一项。

首先我们看一下这个表单的纯 HTML 代码：

文本字段

name、email 和 website 属于文本输入元素，comment 字段是文本框。HTML 代码是这样的：

```
Name: <input type="text" name="name">
E-mail: <input type="text" name="email">
Website: <input type="text" name="website">
Comment: <textarea name="comment" rows="5" cols="40"></textarea>
```

单选按钮

gender 字段是单选按钮，HTML 代码是这样的：

```
Gender:
<input type="radio" name="gender" value="female">Female
<input type="radio" name="gender" value="male">Male
```

表单元素

表单的 HTML 代码是这样的：

```
<form method="post" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
```

当提交此表单时，通过 method="post" 发送表单数据。

什么是 $_SERVER["PHP_SELF"] 变量？

$_SERVER["PHP_SELF"] 是一种超全局变量，它返回当前执行脚本的文件名。

因此，$_SERVER["PHP_SELF"] 将表单数据发送到页面本身，而不是跳转到另一张页面。这样，用户就能够在表单页面获得错误提示信息。

什么是 htmlspecialchars() 函数？

htmlspecialchars() 函数把特殊字符转换为 HTML 实体。这意味着 < 和 > 之类的 HTML 字符会被替换为 < 和 > 。这样可防止攻击者通过在表单中注入 HTML 或 JavaScript 代码（跨站点脚本攻击）对代码进行利用。

关于 PHP 表单安全性的重要提示

$_SERVER["PHP_SELF"] 变量能够被黑客利用！

如果您的页面使用了 PHP_SELF，用户能够输入下划线然后执行跨站点脚本（XSS）。

提示：跨站点脚本（Cross-site scripting，XSS）是一种计算机安全漏洞类型，常见于 Web 应用程序。XSS 能够使攻击者向其他用户浏览的网页中输入客户端脚本。

假设我们的一张名为 "test_form.php" 的页面中有如下表单：

```
<form method="post" action="<?php echo $_SERVER["PHP_SELF"];?>">
```

现在，如果用户进入的是地址栏中正常的 URL："http://www.example.com/test_form.php"，上面的代码会转换为：

```
<form method="post" action="test_form.php">
```

到目前，一切正常。

不过，如果用户在地址栏中键入了如下 URL：

```
http://www.example.com/test_form.php/%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E
```

在这种情况下，上面的代码会转换为：

```
<form method="post" action="test_form.php"/><script>alert('hacked')</script>
```

这段代码加入了一段脚本和一个提示命令。并且当此页面加载后，就会执行 JavaScript 代码（用户会看到一个提示框）。这仅仅是一个关于 PHP_SELF 变量如何被利用的简单无害案例。

您应该意识到 <*script*> 标签内能够添加任何 *JavaScript* 代码！黑客能够把用户重定向到另一台服务器上的某个文件，该文件中的恶意代码能够更改全局变量或将表单提交到其他地址以保存用户数据，等等。

如果避免 $_SERVER["PHP_SELF"] 被利用？

通过使用 htmlspecialchars() 函数能够避免 $_SERVER["PHP_SELF"] 被利用。

表单代码是这样的：

```
<form method="post" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
```

htmlspecialchars() 函数把特殊字符转换为 HTML 实体。现在，如果用户试图利用 PHP_SELF 变量，会导致如下输出：

```
<form method="post" action="test_form.php/"><script>alert('hacked')</script>">
```

无法利用，没有危害！

通过 PHP 验证表单数据

我们要做的第一件事是通过 PHP 的 htmlspecialchars() 函数传递所有变量。

在我们使用 htmlspecialchars() 函数后，如果用户试图在文本字段中提交以下内容：

```
<script>location.href('http://www.hacked.com')</script>
```

- 代码不会执行，因为会被保存为转义代码，就像这样：

```
&lt;script&gt;location.href('http://www.hacked.com')&lt;/script&gt;
```

现在这条代码显示在页面上或 e-mail 中是安全的。

在用户提交该表单时，我们还要做两件事：

1. （通过 PHP trim() 函数）去除用户输入数据中不必要的字符（多余的空格、制表符、换行）
2. （通过 PHP stripslashes() 函数）删除用户输入数据中的反斜杠（\）

接下来我们创建一个检查函数（相比一遍遍地写代码，这样效率更好）。

我们把函数命名为 test_input()。

现在，我们能够通过 test_input() 函数检查每个 $_POST 变量，脚本是这样的：

实例

```
<?php
// 定义变量并设置为空值
$name = $email = $gender = $comment = $website = "";

if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $name = test_input($_POST["name"]);
  $email = test_input($_POST["email"]);
  $website = test_input($_POST["website"]);
  $comment = test_input($_POST["comment"]);
  $gender = test_input($_POST["gender"]);
}

function test_input($data) {
  $data = trim($data);
  $data = stripslashes($data);
  $data = htmlspecialchars($data);
  return $data;
}
?>
```

请注意在脚本开头，我们检查表单是否使用 $_SERVER["REQUEST_METHOD"] 进行提交。如果 REQUEST_METHOD 是 POST，那么表单已被提交 - 并且应该对其进行验证。如果未提交，则跳过验证并显示一个空白表单。

不过，在上面的例子中，所有输入字段都是可选的。即使用户未输入任何数据，脚本也能正常工作。

下一步是制作必填输入字段，并创建需要时使用的错误消息。

PHP 表单验证 - 必填字段

本节展示如何制作必填输入字段，并创建需要时所用的错误消息。

PHP - 输入字段

从上一节中的验证规则中，我们看到 "Name", "E-mail" 以及 "Gender" 字段是必需的。这些字段不能为空且必须在 HTML 表单中填写。

字段	验证规则
Name	必需。必须包含字母和空格。
E-mail	必需。必须包含有效的电子邮件地址（包含 @ 和 .）。
Website	可选。如果选填，则必须包含有效的 URL。
Comment	可选。多行输入字段（文本框）。
Gender	必需。必须选择一项。

在上一节中，所有输入字段都是可选的。

在下面的代码中我们增加了一些新变量：$nameErr、$emailErr、$genderErr 以及 $websiteErr。这些错误变量会保存被请求字段的错误消息。我们还为每个 $_POST 变量添加了一个 if else 语句。这条语句检查 $_POST 变量是否为空（通过 PHP empty() 函数）。如果为空，则错误消息会存储于不同的错误变量中。如果不为空，则通过 test_input() 函数发送用户输入数据：

```
<?php
// 定义变量并设置为空值
$nameErr = $emailErr = $genderErr = $websiteErr = "";
$name = $email = $gender = $comment = $website = "";

if ($_SERVER["REQUEST_METHOD"] == "POST") {
  if (empty($_POST["name"])) {
    $nameErr = "Name is required";
  } else {
    $name = test_input($_POST["name"]);
  }

  if (empty($_POST["email"])) {
    $emailErr = "Email is required";
  } else {
    $email = test_input($_POST["email"]);
  }

  if (empty($_POST["website"])) {
    $website = "";
  } else {
    $website = test_input($_POST["website"]);
  }

  if (empty($_POST["comment"])) {
    $comment = "";
  } else {
    $comment = test_input($_POST["comment"]);
  }

  if (empty($_POST["gender"])) {
    $genderErr = "Gender is required";
  } else {
    $gender = test_input($_POST["gender"]);
  }
}
?>
```

PHP - 显示错误消息

在 HTML 表单中，我们在每个被请求字段后面增加了一点脚本。如果需要，会生成恰当的错误消息（如果用户未填写必填字段就试图提交表单）：

实例

```
<form method="post" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">

Name: <input type="text" name="name">
<span class="error">* <?php echo $nameErr;?></span>
<br><br>
E-mail:
<input type="text" name="email">
<span class="error">* <?php echo $emailErr;?></span>
<br><br>
Website:
<input type="text" name="website">
<span class="error"><?php echo $websiteErr;?></span>
<br><br>
<label>Comment: <textarea name="comment" rows="5" cols="40"></textarea>
<br><br>
Gender:
<input type="radio" name="gender" value="female">Female
<input type="radio" name="gender" value="male">Male
<span class="error">* <?php echo $genderErr;?></span>
<br><br>
<input type="submit" name="submit" value="Submit">

</form>
```

接下来是验证输入数据，即“Name 字段是否只包含字母和空格？”，以及“E-mail 字段是否包含有效的电子邮件地址语法？”，并且如果填写了 Website 字段，“这个字段是否包含了有效的 URL？”。

PHP 表单验证 - 验证 E-mail 和 URL

本节展示如何验证名字、电邮和 URL。

PHP - 验证名字

以下代码展示的简单方法检查 name 字段是否包含字母和空格。如果 name 字段无效，则存储一条错误消息：

```
$name = test_input($_POST["name"]);
if (!preg_match("/^[a-zA-Z ]*$/",$name)) {
  $nameErr = "只允许字母和空格！";
}
```

注释：preg_match() 函数检索字符串的模式，如果模式存在则返回 true，否则返回 false。

PHP - 验证 E-mail

以下代码展示的简单方法检查 e-mail 地址语法是否有效。如果无效则存储一条错误消息：

```
$email = test_input($_POST["email"]);
if (!preg_match("/([\w\-]+\@[\w\-]+\.[\w\-]+)/",$email)) {
  $emailErr = "无效的 email 格式！";
}
```

PHP - 验证 URL

以下代码展示的方法检查 URL 地址语法是否有效（这条正则表达式同时允许 URL 中的斜杠）。如果 URL 地址语法无效，则存储一条错误消息：

```
$website = test_input($_POST["website"]);
if (!preg_match("/\b(?:(?:https?|ftp):\/\/|www\.)[-a-z0-9+&@#\/%?=~_|!:,.;]*[-a-z0-9+&@#\
=~_|]/i",$website)) {
  $websiteErr = "无效的 URL";
}
```

PHP - 验证 Name、E-mail、以及 URL

现在，脚本是这样的：

实例

```
<?php
// 定义变量并设置为空值
$nameErr = $emailErr = $genderErr = $websiteErr = "";
$name = $email = $gender = $comment = $website = "";

if ($_SERVER["REQUEST_METHOD"] == "POST") {
  if (empty($_POST["name"])) {
    $nameErr = "Name is required";
  } else {
    $name = test_input($_POST["name"]);
    // 检查名字是否包含字母和空格
    if (!preg_match("/^[a-zA-Z ]*$/",$name)) {
      $nameErr = "Only letters and white space allowed";
    }
  }

  if (empty($_POST["email"])) {
    $emailErr = "Email is required";
  } else {
    $email = test_input($_POST["email"]);
    // 检查电邮地址语法是否有效
    if (!preg_match("/([\w\-]+\@[\w\-]+\.[\w\-]+)/",$email)) {
      $emailErr = "Invalid email format";
    }
  }

  if (empty($_POST["website"])) {
    $website = "";
  } else {
    $website = test_input($_POST["website"]);
    // 检查 URL 地址语言是否有效（此正则表达式同样允许 URL 中的下划线）
    if (!preg_match("/\b(?:(?:https?|ftp):\/\/|www\.)[-a-z0-9+&@#\/%?=~_|!:,.;]*[-a-z0-9+
    =~_|]/i",$website)) {
      $websiteErr = "Invalid URL";
    }
  }

  if (empty($_POST["comment"])) {
    $comment = "";
  } else {
    $comment = test_input($_POST["comment"]);
  }

  if (empty($_POST["gender"])) {
    $genderErr = "Gender is required";
  } else {
    $gender = test_input($_POST["gender"]);
  }
}
?>
```

接下来向您讲解如何防止表单在用户提交表单后清空所有输入字段。

PHP 表单验证 - 完成表单实例

本节展示如何在用户提交表单后保留输入字段中的值。

PHP - 保留表单中的值

如需在用户点击提交按钮后在输入字段中显示值，我们在以下输入字段的 value 属性中增加了一小段 PHP 脚本：name、email 以及 website。在 comment 文本框字段中，我们把脚本放到了 <textarea> 与 </textarea> 之间。这些脚本输出 $name、$email、$website 和 $comment 变量的值。

然后，我们还需要显示选中了哪个单选按钮。对此，我们必须操作 checked 属性（而非单选按钮的 value 属性）：

```
Name: <input type="text" name="name" value="<?php echo $name;?>">

E-mail: <input type="text" name="email" value="<?php echo $email;?>">

Website: <input type="text" name="website" value="<?php echo $website;?>">

Comment: <textarea name="comment" rows="5" cols="40"><?php echo $comment;?></textarea>

Gender:

<input type="radio" name="gender"
<?php if (isset($gender) && $gender=="female") echo "checked";?>
value="female">Female
<input type="radio" name="gender"
<?php if (isset($gender) && $gender=="male") echo "checked";?>
value="male">Male
```

PHP - 完整的表单实例

下面是 PHP 表单验证实例的完整代码：

实例

```
<h2>PHP 表单验证实例</h2>
<p><span class="error">* 必填字段</span></p>

<form method="post" action="/demo/demo_form_validation_complete.php">

姓名：
<input type="text" name="name" value="">
<span class="error">* </span>
<br><br>
电邮：
<input type="text" name="email" value="">
<span class="error">* </span>
<br><br>
网址：
<input type="text" name="website" value="">
<span class="error"></span>
<br><br>
<label>
评论：
<textarea name="comment" rows="5" cols="40"></textarea>
<br><br>
性别：
<input type="radio" name="gender"  value="female">女性
<input type="radio" name="gender"  value="male">男性
<span class="error">* </span>
<br><br>
<input type="submit" name="submit" value="提交">

</form>

<h2>您的输入：</h2>
```

PHP $_GET 变量

在 PHP 中，预定义的 $_GET 变量用于收集来自 method="get" 的表单中的值。

$_GET 变量

预定义的 $_GET 变量用于收集来自 method="get" 的表单中的值。

从带有 GET 方法的表单发送的信息，对任何人都是可见的（会显示在浏览器的地址栏），并且对发送信息的量也有限制。

实例

```
<form action="welcome.php" method="get">
Name: <input type="text" name="fname">
Age: <input type="text" name="age">
<input type="submit">
</form>
```

当用户点击 "Submit" 按钮时，发送到服务器的 URL 如下所示：

http://www.w3cschool.cc/welcome.php?fname=Peter&age=37

"welcome.php" 文件现在可以通过 $_GET 变量来收集表单数据了（请注意，表单域的名称会自动成为 $_GET 数组中的键）：

```
Welcome <?php echo $_GET["fname"]; ?>.<br>
You are <?php echo $_GET["age"]; ?> years old!
```

何时使用 method="get"？

在 HTML 表单中使用 method="get" 时，所有的变量名和值都会显示在 URL 中。

注释：所以在发送密码或其他敏感信息时，不应该使用这个方法！

然而，正因为变量显示在 URL 中，因此可以在收藏夹中收藏该页面。在某些情况下，这是很有用的。

注释：HTTP GET 方法不适合大型的变量值。它的值是不能超过 2000 个字符的。

PHP $_POST 变量

在 PHP 中，预定义的 $_POST 变量用于收集来自 method="post" 的表单中的值。

$_POST 变量

预定义的 $_POST 变量用于收集来自 method="post" 的表单中的值。

从带有 POST 方法的表单发送的信息，对任何人都是不可见的（不会显示在浏览器的地址栏），并且对发送信息的量也没有限制。

注释：然而，默认情况下，POST 方法的发送信息的量最大值为 8 MB（可通过设置 php.ini 文件中的 post_max_size 进行更改）。

实例

```
<form action="welcome.php" method="post">
Name: <input type="text" name="fname">
Age: <input type="text" name="age">
<input type="submit">
</form>
```

当用户点击 "Submit" 按钮时，URL 如下所示：

http://www.w3cschool.cc/welcome.php

"welcome.php" 文件现在可以通过 $_POST 变量来收集表单数据了（请注意，表单域的名称会自动成为 $_POST 数组中的键）：

```
Welcome <?php echo $_POST["fname"]; ?>!<br>
You are <?php echo $_POST["age"]; ?> years old.
```

何时使用 method="post"？

从带有 POST 方法的表单发送的信息，对任何人都是不可见的，并且对发送信息的量也没有限制。

然而，由于变量不显示在 URL 中，所以无法把页面加入书签。

PHP $_REQUEST 变量

预定义的 $_REQUEST 变量包含了 $_GET、$_POST 和 $_COOKIE 的内容。

$_REQUEST 变量可用来收集通过 GET 和 POST 方法发送的表单数据。

实例

```
Welcome <?php echo $_REQUEST["fname"]; ?>!<br>
You are <?php echo $_REQUEST["age"]; ?> years old.
```

PHP 高级

PHP 多维数组

在本教程之前的章节中，我们已经知道数组是一种数/值对的简单列表。

不过，有时您希望用一个以上的键存储值。

可以用多维数组进行存储。

PHP - 多维数组

多维数组指的是包含一个或多个数组的数组。

PHP 能理解两、三、四或五级甚至更多级的多维数组。不过，超过三级深的数组对于大多数人难于管理。

注释：数组的维度指示您需要选择元素的索引数。

- 对于二维数组，您需要两个索引来选取元素
- 对于三维数组，您需要三个索引来选取元素

PHP - 两维数组

两维数组是数组的数组（三维数组是数组的数组的数组）。

首先，让我们看看下面的表格：

品牌	库存	销量
Volvo	33	20
BMW	17	15
Saab	5	2
Land Rover	15	11

我们能够在两维数组中存储上表中的数据，就像这样：

```
$cars = array
  (
  array("Volvo",22,18),
  array("BMW",15,13),
  array("Saab",5,2),
  array("Land Rover",17,15)
  );
```

现在这个两维数组包含了四个数组，并且它有两个索引（下标）：行和列。

如需访问 $cars 数组中的元素，我们必须使用两个索引（行和列）：

实例

```
<?php
echo $cars[0][0].": 库存：".$cars[0][1].", 销量：".$cars[0][2].".<br>";
echo $cars[1][0].": 库存：".$cars[1][1].", 销量：".$cars[1][2].".<br>";
echo $cars[2][0].": 库存：".$cars[2][1].", 销量：".$cars[2][2].".<br>";
echo $cars[3][0].": 库存：".$cars[3][1].", 销量：".$cars[3][2].".<br>";
?>
```

我们也可以在 For 循环中使用另一个 For 循环，来获得 $cars 数组中的元素（我们仍需使用两个索引）：

实例

```
<?php
for ($row = 0; $row < 4; $row++) {
  echo "<p><b>Row number $row</b></p>";
  echo "<ul>";
  for ($col = 0; $col < 3; $col++) {
    echo "<li>".$cars[$row][$col]."</li>";
  }
  echo "</ul>";
}
?>
```

PHP 日期和时间

PHP date() 函数用于对日期或时间进行格式化。

PHP Date() 函数

PHP Date() 函数把时间戳格式化为更易读的日期和时间。

语法

```
date(format,timestamp)
```

参数	描述
format	必需。规定时间戳的格式。
timestamp	可选。规定时间戳。默认是当前时间和日期。

注释：时间戳是一种字符序列，它表示具体事件发生的日期和事件。

获得简单的日期

date() 函数的格式参数是必需的，它们规定如何格式化日期或时间。

下面列出了一些常用于日期的字符：

- d - 表示月里的某天（01-31）
- m - 表示月（01-12）
- Y - 表示年（四位数）
- 1 - 表示周里的某天

其他字符，比如 "/", "." 或 "-" 也可被插入字符中，以增加其他格式。

下面的例子用三种不同方法格式今天的日期：

实例

```
<?php
echo "今天是 " . date("Y/m/d") . "<br>";
echo "今天是 " . date("Y.m.d") . "<br>";
echo "今天是 " . date("Y-m-d") . "<br>";
echo "今天是 " . date("l");
?>
```

PHP 提示 - 自动版权年份

使用 date() 函数在您的网站上自动更新版本年份：

实例

```
? 2010-<?php echo date("Y")?>
```

获得简单的时间

下面是常用于时间的字符：

- h - 带有首位零的 12 小时小时格式
- i - 带有首位零的分钟
- s - 带有首位零的秒（00 -59）
- a - 小写的午前和午后（am 或 pm）

下面的例子以指定的格式输出当前时间：

实例

```
<?php
echo "现在时间是 " . date("h:i:sa");
?>
```

注释：请注意 PHP date() 函数会返回服务器的当前日期/时间！

获得时区

如果从代码返回的不是正确的时间，有可能是因为您的服务器位于其他国家或者被设置为不同时区。

因此，如果您需要基于具体位置的准确时间，您可以设置要用的时区。

下面的例子把时区设置为 "Asia/Shanghai"，然后以指定格式输出当前时间：

实例

```
<?php
date_default_timezone_set("Asia/Shanghai");
echo "当前时间是 " . date("h:i:sa");
?>
```

通过 PHP mktime() 创建日期

date() 函数中可选的时间戳参数规定时间戳。如果您未规定时间戳，将使用当前日期和时间（正如上例中那样）。

mktime() 函数返回日期的 Unix 时间戳。Unix 时间戳包含 Unix 纪元（1970 年 1 月 1 日 00:00:00 GMT）与指定时间之间的秒数。

语法

```
mktime(hour,minute,second,month,day,year)
```

下面的例子使用 mktime() 函数中的一系列参数来创建日期和时间：

实例

```
<?php
$d=mktime(9, 12, 31, 6, 10, 2015);
echo "创建日期是 " . date("Y-m-d h:i:sa", $d);
?>
```

通过 PHP strtotime() 用字符串来创建日期

PHP strtotime() 函数用于把人类可读的字符串转换为 Unix 时间。

语法

```
strtotime(time,now)
```

下面的例子通过 strtotime() 函数创建日期和时间：

实例

```
<?php
$d=strtotime("10:38pm April 15 2015");
echo "创建日期是 " . date("Y-m-d h:i:sa", $d);
?>
```

PHP 在将字符串转换为日期这方面非常聪明，所以您能够使用各种值：

实例

```
<?php
$d=strtotime("tomorrow");
echo date("Y-m-d h:i:sa", $d) . "<br>";

$d=strtotime("next Saturday");
echo date("Y-m-d h:i:sa", $d) . "<br>";

$d=strtotime("+3 Months");
echo date("Y-m-d h:i:sa", $d) . "<br>";
?>
```

不过，strtotime() 并不完美，所以请记得检查放入其中的字符串。

更多日期实例

下例输出下周六的日期：

实例

```
<?php
$startdate = strtotime("Saturday");
$enddate = strtotime("+6 weeks",$startdate);

while ($startdate < $enddate) {
  echo date("M d", $startdate),"<br>";
  $startdate = strtotime("+1 week", $startdate);
}
?>
```

下例输出七月四日之前的天数：

实例

```
<?php
$d1=strtotime("December 31");
$d2=ceil(($d1-time())/60/60/24);
echo "距离十二月三十一日还有：" . $d2 ." 天。";
?>
```

完整的 PHP 日期参考手册

如需所有日期函数的完整手册，请访问我们的 PHP 日期参考手册。

该手册包含每个函数的简要描述以及使用示例。

PHP Include 文件

服务器端包含 **(SSI)** 用于创建可在多个页面重复使用的函数、页眉、页脚或元素。

include （或 require）语句会获取指定文件中存在的所有文本/代码/标记，并复制到使用 include 语句的文件中。

包含文件很有用，如果您需要在网站的多张页面上引用相同的 PHP、HTML 或文本的话。

PHP include 和 require 语句

通过 include 或 require 语句，可以将 PHP 文件的内容插入另一个 PHP 文件（在服务器执行它之前）。

include 和 require 语句是相同的，除了错误处理方面：

- require 会生成致命错误（E_COMPILE_ERROR）并停止脚本
- include 只生成警告（E_WARNING），并且脚本会继续

因此，如果您希望继续执行，并向用户输出结果，即使包含文件已丢失，那么请使用 include。否则，在框架、CMS 或者复杂的 PHP 应用程序编程中，请始终使用 require 向执行流引用关键文件。这有助于提高应用程序的安全性和完整性，在某个关键文件意外丢失的情况下。

包含文件省去了大量的工作。这意味着您可以为所有页面创建标准页头、页脚或者菜单文件。然后，在页头需要更新时，您只需更新这个页头包含文件即可。

语法

```
include 'filename';
```

或

```
require 'filename';
```

PHP include 实例

例子 1

假设我们有一个名为 "footer.php" 的标准的页脚文件，就像这样：

```
<?php
echo "<p>Copyright ? 2006-" . date("Y") . " W3School.com.cn</p>";
?>
```

如需在一张页面中引用这个页脚文件，请使用 include 语句：

```
<html>
<body>

<h1>欢迎访问我们的首页！</h1>
<p>一段文本。</p>
<p>一段文本。</p>
<?php include 'footer.php';?>

</body>
</html>
```

例子 2

假设我们有一个名为 "menu.php" 的标准菜单文件：

```
<?php
echo '<a href="/index.asp">首页</a> -
<a href="/html/index.asp">HTML 教程</a> -
<a href="/css/index.asp">CSS 教程</a> -
<a href="/js/index.asp">JavaScript 教程</a> -
<a href="/php/index.asp">PHP 教程</a>';
?>
```

网站中的所有页面均使用此菜单文件。具体的做法是（我们使用了一个 <div> 元素，这样今后就可以轻松地通过 CSS 设置样式）：

```
<html>
<body>

<div class="menu">
<?php include 'menu.php';?>
</div>

<h1>欢迎访问我的首页！</h1>
<p>Some text.</p>
<p>Some more text.</p>

</body>
</html>
```

例子 3

假设我们有一个名为 "vars.php" 的文件，其中定义了一些变量：

```
<?php
$color='银色的';
$car='奔驰轿车';
?>
```

然后，如果我们引用这个 "vars.php" 文件，就可以在调用文件中使用这些变量：

```
<html>
<body>

<h1>欢迎访问我的首页！</h1>
<?php
include 'vars.php';
echo "我有一辆" . $color . $car "。";
?>

</body>
</html>
```

PHP include vs. require

require 语句同样用于向 PHP 代码中引用文件。

不过，include 与 require 有一个巨大的差异：如果用 include 语句引用某个文件并且 PHP 无法找到它，脚本会继续执行：

实例

```
<html>
<body>

<h1>Welcome to my home page!</h1>
<?php
include 'noFileExists.php';
echo "I have a $color $car.";
?>

</body>
</html>
```

如果我们使用 require 语句完成相同的案例，echo 语句不会继续执行，因为在 require 语句返回严重错误之后脚本就会终止执行：

实例

```
<html>
<body>

<h1>Welcome to my home page!</h1>
<?php
require 'noFileExists.php';
echo "I have a $color $car.";
?>

</body>
</html>
```

注释：

请在此时使用 require：当文件被应用程序请求时。

请在此时使用 include：当文件不是必需的，且应用程序在文件未找到时应该继续运行时。

PHP 文件处理

PHP 操作文件

PHP 拥有的多种函数可供创建、读取、上传以及编辑文件。

注意：请谨慎操作文件！

当您操作文件时必须非常小心。如果您操作失误，可能会造成非常严重的破坏。常见的错误是：

- 编辑错误的文件
- 被垃圾数据填满硬盘
- 意外删除文件内容

PHP readfile() 函数

readfile() 函数读取文件，并把它写入输出缓冲。

假设我们有一个名为 "webdictionary.txt" 的文本文件，存放在服务器上，就像这样：

```
AJAX = Asynchronous JavaScript and XML
CSS = Cascading Style Sheets
HTML = Hyper Text Markup Language
PHP = PHP Hypertext Preprocessor
SQL = Structured Query Language
SVG = Scalable Vector Graphics
XML = EXtensible Markup Language
```

读取此文件并写到输出流的 PHP 代码如下（如读取成功则 readfile() 函数返回字节数）：

实例

```
<?php
echo readfile("webdictionary.txt");
?>
```

如果您想做的所有事情就是打开一个文件并读取器内容，那么 readfile() 函数很有用。

下一节会讲解更多有关文件处理的内容。

PHP Filesystem 参考手册

如需完整的 PHP 文件系统参考手册，请访问 W3School 提供的 PHP Filesystem 参考手册。

PHP 文件打开/读取/读取

在本节中，我们向您讲解如何在服务器上打开、读取以及关闭文件。

PHP Open File - fopen()

打开文件的更好的方法是通过 fopen() 函数。此函数为您提供比 readfile() 函数更多的选项。

在课程中，我们将使用文本文件 "webdictionary.txt"：

```
AJAX = Asynchronous JavaScript and XML
CSS = Cascading Style Sheets
HTML = Hyper Text Markup Language
PHP = PHP Hypertext Preprocessor
SQL = Structured Query Language
SVG = Scalable Vector Graphics
XML = EXtensible Markup Language
```

fopen() 的第一个参数包含被打开的文件名，第二个参数规定打开文件的模式。如果 fopen() 函数未能打开指定的文件，下面的例子会生成一段消息：

实例

```
<?php
$myfile = fopen("webdictionary.txt", "r") or die("Unable to open file!");
echo fread($myfile,filesize("webdictionary.txt"));
fclose($myfile);
?>
```

提示：我们接下来将学习 fread() 以及 fclose() 函数。

文件会以如下模式之一打开：

模式	描述
r	打开文件为只读。文件指针在文件的开头开始。
w	打开文件为只写。删除文件的内容或创建一个新的文件，如果它不存在。文件指针在文件的开头开始。
a	打开文件为只写。文件中的现有数据会被保留。文件指针在文件结尾开始。创建新的文件，如果文件不存在。
x	创建新文件为只写。返回 FALSE 和错误，如果文件已存在。
r+	打开文件为读/写、文件指针在文件开头开始。
w+	打开文件为读/写。删除文件内容或创建新文件，如果它不存在。文件指针在文件开头开始。
a+	打开文件为读/写。文件中已有的数据会被保留。文件指针在文件结尾开始。创建新文件，如果它不存在。
x+	创建新文件为读/写。返回 FALSE 和错误，如果文件已存在。

PHP 读取文件 - fread()

fread() 函数读取打开的文件。

fread() 的第一个参数包含待读取文件的文件名，第二个参数规定待读取的最大字节数。

如下 PHP 代码把 "webdictionary.txt" 文件读至结尾：

```
fread($myfile,filesize("webdictionary.txt"));
```

PHP 关闭文件 - fclose()

fclose() 函数用于关闭打开的文件。

注释：用完文件后把它们全部关闭是一个良好的编程习惯。您并不想打开的文件占用您的服务器资源。

fclose() 需要待关闭文件的名称（或者存有文件名的变量）：

```
<?php
$myfile = fopen("webdictionary.txt", "r");
// some code to be executed....
fclose($myfile);
?>
```

PHP 读取单行文件 - fgets()

fgets() 函数用于从文件读取单行。

下例输出 "webdictionary.txt" 文件的首行：

实例

```
<?php
$myfile = fopen("webdictionary.txt", "r") or die("Unable to open file!");
echo fgets($myfile);
fclose($myfile);
?>
```

注释：调用 fgets() 函数之后，文件指针会移动到下一行。

PHP 检查 End-Of-File - feof()

feof() 函数检查是否已到达 "end-of-file" (EOF)。

feof() 对于遍历未知长度的数据很有用。

下例逐行读取 "webdictionary.txt" 文件，直到 end-of-file：

实例

```
<?php
$myfile = fopen("webdictionary.txt", "r") or die("Unable to open file!");
// 输出单行直到 end-of-file
while(!feof($myfile)) {
  echo fgets($myfile) . "<br>";
}
fclose($myfile);
?>
```

PHP 读取单字符 - fgetc()

fgetc() 函数用于从文件中读取单个字符。

下例逐字符读取 "webdictionary.txt" 文件，直到 end-of-file：

实例

```
<?php
$myfile = fopen("webdictionary.txt", "r") or die("Unable to open file!");
// 输出单字符直到 end-of-file
while(!feof($myfile)) {
  echo fgetc($myfile);
}
fclose($myfile);
?>
```

注释：在调用 fgetc() 函数之后，文件指针会移动到下一个字符。

PHP Filesystem 参考手册

如需完整的 PHP 文件系统参考手册，请访问 W3School 提供的 PHP Filesystem 参考手册。

PHP 文件创建/写入

在本节中，我们将为您讲解如何在服务器上创建并写入文件。

PHP 创建文件 - fopen()

fopen() 函数也用于创建文件。也许有点混乱，但是在 PHP 中，创建文件所用的函数与打开文件的相同。

如果您用 fopen() 打开并不存在的文件，此函数会创建文件，假定文件被打开为写入（w）或增加（a）。

下面的例子创建名为 "testfile.txt" 的新文件。此文件将被创建于 PHP 代码所在的相同目录中：

实例

```
$myfile = fopen("testfile.txt", "w")
```

PHP 文件权限

如果您试图运行这段代码时发生错误，请检查您是否有向硬盘写入信息的 PHP 文件访问权限。

PHP 写入文件 - fwrite()

fwrite() 函数用于写入文件。

fwrite() 的第一个参数包含要写入的文件的文件名，第二个参数是被写的字符串。

下面的例子把姓名写入名为 "newfile.txt" 的新文件中：

实例

```
<?php
$myfile = fopen("newfile.txt", "w") or die("Unable to open file!");
$txt = "Bill Gates\n";
fwrite($myfile, $txt);
$txt = "Steve Jobs\n";
fwrite($myfile, $txt);
fclose($myfile);
?>
```

请注意，我们向文件 "newfile.txt" 写了两次。在每次我们向文件写入时，在我们发送的字符串 $txt 中，第一次包含 "Bill Gates"，第二次包含 "Steve Jobs"。在写入完成后，我们使用 fclose() 函数来关闭文件。

如果我们打开 "newfile.txt" 文件，它应该是这样的：

```
Bill Gates
Steve Jobs
```

PHP 覆盖（Overwriting）

如果现在 "newfile.txt" 包含了一些数据，我们可以展示在写入已有文件时发生的的事情。所有已存在的数据会被擦除并以一个新文件开始。

在下面的例子中，我们打开一个已存在的文件 "newfile.txt"，并向其中写入了一些新数据：

实例

```
<?php
$myfile = fopen("newfile.txt", "w") or die("Unable to open file!");
$txt = "Mickey Mouse\n";
fwrite($myfile, $txt);
$txt = "Minnie Mouse\n";
fwrite($myfile, $txt);
fclose($myfile);
?>
```

如果现在我们打开这个 "newfile.txt" 文件，Bill 和 Steve 都已消失，只剩下我们刚写入的数据：

```
Mickey Mouse
Minnie Mouse
```

PHP Filesystem 参考手册

如需完整的 PHP 文件系统参考手册，请访问 W3School 提供的 PHP Filesystem 参考手册。

PHP 文件上传

通过 **PHP**，可以把文件上传到服务器。

创建一个文件上传表单

允许用户从表单上传文件是非常有用的。

请看下面这个供上传文件的 HTML 表单：

```
<html>
<body>

<form action="upload_file.php" method="post"
enctype="multipart/form-data">
<label for="file">Filename:</label>
<input type="file" name="file" id="file" />
<br />
<input type="submit" name="submit" value="Submit" />
</form>

</body>
</html>
```

请留意如下有关此表单的信息：

<form> 标签的 enctype 属性规定了在提交表单时要使用哪种内容类型。在表单需要二进制数据时，比如文件内容，请使用 "multipart/form-data"。

<input> 标签的 type="file" 属性规定了应该把输入作为文件来处理。举例来说，当在浏览器中预览时，会看到输入框旁边有一个浏览按钮。

注释：允许用户上传文件是一个巨大的安全风险。请仅仅允许可信的用户执行文件上传操作。

创建上传脚本

"upload_file.php" 文件含有供上传文件的代码：

```
<?php
if ($_FILES["file"]["error"] > 0)
  {
  echo "Error: " . $_FILES["file"]["error"] . "<br />";
  }
else
  {
  echo "Upload: " . $_FILES["file"]["name"] . "<br />";
  echo "Type: " . $_FILES["file"]["type"] . "<br />";
  echo "Size: " . ($_FILES["file"]["size"] / 1024) . " Kb<br />";
  echo "Stored in: " . $_FILES["file"]["tmp_name"];
  }
?>
```

通过使用 PHP 的全局数组 $_FILES，你可以从客户计算机向远程服务器上传文件。

第一个参数是表单的 input name，第二个下标可以是 "name", "type", "size", "tmp_name" 或 "error"。就像这样：

- $_FILES["file"]["name"] - 被上传文件的名称
- $_FILES["file"]["type"] - 被上传文件的类型
- $_FILES["file"]["size"] - 被上传文件的大小，以字节计
- $_FILES["file"]["tmp_name"] - 存储在服务器的文件的临时副本的名称
- $_FILES["file"]["error"] - 由文件上传导致的错误代码

这是一种非常简单文件上传方式。基于安全方面的考虑，您应当增加有关什么用户有权上传文件的限制。

上传限制

在这个脚本中，我们增加了对文件上传的限制。用户只能上传 .gif 或 .jpeg 文件，文件大小必须小于 20 kb：

```
<?php

if ((($_FILES["file"]["type"] == "image/gif")
|| ($_FILES["file"]["type"] == "image/jpeg")
|| ($_FILES["file"]["type"] == "image/pjpeg"))
&& ($_FILES["file"]["size"] < 20000))
  {
  if ($_FILES["file"]["error"] > 0)
    {
    echo "Error: " . $_FILES["file"]["error"] . "<br />";
    }
  else
    {
    echo "Upload: " . $_FILES["file"]["name"] . "<br />";
    echo "Type: " . $_FILES["file"]["type"] . "<br />";
    echo "Size: " . ($_FILES["file"]["size"] / 1024) . " Kb<br />";
    echo "Stored in: " . $_FILES["file"]["tmp_name"];
    }
  }
else
  {
  echo "Invalid file";
  }

?>
```

注释：对于 IE，识别 jpg 文件的类型必须是 pjpeg，对于 FireFox，必须是 jpeg。

保存被上传的文件

上面的例子在服务器的 PHP 临时文件夹创建了一个被上传文件的临时副本。

这个临时的复制文件会在脚本结束时消失。要保存被上传的文件，我们需要把它拷贝到另外的位置：

```
<?php
if ((($_FILES["file"]["type"] == "image/gif")
|| ($_FILES["file"]["type"] == "image/jpeg")
|| ($_FILES["file"]["type"] == "image/pjpeg"))
&& ($_FILES["file"]["size"] < 20000))
  {
  if ($_FILES["file"]["error"] > 0)
    {
    echo "Return Code: " . $_FILES["file"]["error"] . "<br />";
    }
  else
    {
    echo "Upload: " . $_FILES["file"]["name"] . "<br />";
    echo "Type: " . $_FILES["file"]["type"] . "<br />";
    echo "Size: " . ($_FILES["file"]["size"] / 1024) . " Kb<br />";
    echo "Temp file: " . $_FILES["file"]["tmp_name"] . "<br />";

    if (file_exists("upload/" . $_FILES["file"]["name"]))
      {
      echo $_FILES["file"]["name"] . " already exists. ";
      }
    else
      {
      move_uploaded_file($_FILES["file"]["tmp_name"],
      "upload/" . $_FILES["file"]["name"]);
      echo "Stored in: " . "upload/" . $_FILES["file"]["name"];
      }
    }
  }
else
  {
  echo "Invalid file";
  }
?>
```

上面的脚本检测了是否已存在此文件，如果不存在，则把文件拷贝到指定的文件夹。

注释：这个例子把文件保存到了名为 "upload" 的新文件夹。

PHP Cookies

cookie 常用于识别用户。

什么是 Cookie？

cookie 常用于识别用户。cookie 是服务器留在用户计算机中的小文件。每当相同的计算机通过浏览器请求页面时，它同时会发送 cookie。通过 PHP，您能够创建并取回 cookie 的值。

如何创建 cookie？

setcookie() 函数用于设置 cookie。

注释：setcookie() 函数必须位于 <html> 标签之前。

语法

```
setcookie(name, value, expire, path, domain);
```

例子

在下面的例子中，我们将创建名为 "user" 的 cookie，把为它赋值 "Alex Porter"。我们也规定了此 cookie 在一小时后过期：

```
<?php
setcookie("user", "Alex Porter", time()+3600);
?>

<html>
<body>

</body>
</html>
```

注释：在发送 cookie 时，cookie 的值会自动进行 URL 编码，在取回时进行自动解码（为防止 URL 编码，请使用 setrawcookie() 取而代之）。

如何取回 Cookie 的值？

PHP 的 $_COOKIE 变量用于取回 cookie 的值。

在下面的例子中，我们取回了名为 "user" 的 cookie 的值，并把它显示在了页面上：

```
<?php
// Print a cookie
echo $_COOKIE["user"];

// A way to view all cookies
print_r($_COOKIE);
?>
```

在下面的例子中，我们使用 isset() 函数来确认是否已设置了 cookie：

```
<html>
<body>

<?php
if (isset($_COOKIE["user"]))
  echo "Welcome " . $_COOKIE["user"] . "!<br />";
else
  echo "Welcome guest!<br />";
?>

</body>
</html>
```

如何删除 cookie？

当删除 cookie 时，您应当使过期日期变更为过去的时间点。

删除的例子：

```
<?php
// set the expiration date to one hour ago
setcookie("user", "", time()-3600);
?>
```

如果浏览器不支持 cookie 该怎么办？

如果您的应用程序涉及不支持 cookie 的浏览器，您就不得不采取其他方法在应用程序中从一张页面向另一张页面传递信息。一种方式是从表单传递数据（有关表单和用户输入的内容，稍早前我们已经在本教程中介绍过了）。

下面的表单在用户单击提交按钮时向 "welcome.php" 提交了用户输入：

```
<html>
<body>

<form action="welcome.php" method="post">
Name: <input type="text" name="name" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>

</body>
</html>
```

取回 "welcome.php" 中的值，就像这样：

```
<html>
<body>

Welcome <?php echo $_POST["name"]; ?>.<br />
You are <?php echo $_POST["age"]; ?> years old.

</body>
</html>
```

PHP Sessions

PHP session 变量用于存储有关用户会话的信息，或更改用户会话的设置。**Session** 变量保存的信息是单一用户的，并且可供应用程序中的所有页面使用。

PHP Session 变量

当您运行一个应用程序时，您会打开它，做些更改，然后关闭它。这很像一次会话。计算机清楚你是谁。它知道你何时启动应用程序，并在何时终止。但是在因特网上，存在一个问题：服务器不知道你是谁以及你做什么，这是由于 HTTP 地址不能维持状态。

通过在服务器上存储用户信息以便随后使用，PHP session 解决了这个问题（比如用户名称、购买商品等）。不过，会话信息是临时的，在用户离开网站后将被删除。如果您需要永久储存信息，可以把数据存储在数据库中。

Session 的工作机制是：为每个访问者创建一个唯一的 id (UID)，并基于这个 UID 来存储变量。UID 存储在 cookie 中，亦或通过 URL 进行传导。

开始 PHP Session

在您把用户信息存储到 PHP session 中之前，首先必须启动会话。

注释：session_start() 函数必须位于 <html> 标签之前：

```
<?php session_start(); ?>

<html>
<body>

</body>
</html>
```

上面的代码会向服务器注册用户的会话，以便您可以开始保存用户信息，同时会为用户会话分配一个 UID。

存储 Session 变量

存储和取回 session 变量的正确方法是使用 PHP $_SESSION 变量：

```
<?php
session_start();
// store session data
$_SESSION['views']=1;
?>

<html>
<body>

<?php
//retrieve session data
echo "Pageviews=". $_SESSION['views'];
?>

</body>
</html>
```

输出：

```
Pageviews=1
```

在下面的例子中，我们创建了一个简单的 page-view 计数器。isset() 函数检测是否已设置 "views" 变量。如果已设置 "views" 变量，我们累加计数器。如果 "views" 不存在，则我们创建 "views" 变量，并把它设置为 1：

```
<?php
session_start();

if(isset($_SESSION['views']))
  $_SESSION['views']=$_SESSION['views']+1;

else
  $_SESSION['views']=1;
echo "Views=". $_SESSION['views'];
?>
```

终结 Session

如果您希望删除某些 session 数据，可以使用 unset() 或 session_destroy() 函数。

unset() 函数用于释放指定的 session 变量：

```
<?php
unset($_SESSION['views']);
?>
```

您也可以通过 session_destroy() 函数彻底终结 session：

```
<?php
session_destroy();
?>
```

注释：session_destroy() 将重置 session，您将失去所有已存储的 session 数据。

PHP 发送电子邮件

PHP 允许您从脚本直接发送电子邮件。

PHP mail() 函数

PHP mail() 函数用于从脚本中发送电子邮件。

语法

```
mail(to,subject,message,headers,parameters)
```

参数	描述
to	必需。规定 email 接收者。
subject	必需。规定 email 的主题。注释：该参数不能包含任何新行字符。
message	必需。定义要发送的消息。应使用 LF (\n) 来分隔各行。
headers	可选。规定附加的标题，比如 From、Cc 以及 Bcc。 应当使用 CRLF (\r\n) 分隔附加的标题。
parameters	可选。对邮件发送程序规定额外的参数。

注释：PHP 需要一个已安装且正在运行的邮件系统，以便使邮件函数可用。所用的程序通过在 php.ini 文件中的配置设置进行定义。请在我们的 PHP Mail 参考手册阅读更多内容。

PHP 简易 E-Mail

通过 PHP 发送电子邮件的最简单的方式是发送一封文本 email。

在下面的例子中，我们首先声明变量($to, $subject, $message, $from, $headers)，然后我们在 mail() 函数中使用这些变量来发送了一封 e-mail：

```
<?php

$to = "someone@example.com";
$subject = "Test mail";
$message = "Hello! This is a simple email message.";
$from = "someonelse@example.com";
$headers = "From: $from";
mail($to,$subject,$message,$headers);
echo "Mail Sent.";

?>
```

PHP Mail Form

通过 PHP，您能够在自己的站点制作一个反馈表单。下面的例子向指定的 e-mail 地址发送了一条文本消息：

```
<html>
<body>

<?php
if (isset($_REQUEST['email']))
//if "email" is filled out, send email
  {
  //send email
  $email = $_REQUEST['email'] ;
  $subject = $_REQUEST['subject'] ;
  $message = $_REQUEST['message'] ;
  mail( "someone@example.com", "Subject: $subject",
  $message, "From: $email" );
  echo "Thank you for using our mail form";
  }
else
//if "email" is not filled out, display the form
  {
  echo "<form method='post' action='mailform.php'>
  Email: <input name='email' type='text' /><br />
  Subject: <input name='subject' type='text' /><br />
  Message:<br />
  <textarea name='message' rows='15' cols='40'>
  </textarea><br />
  <input type='submit' />
  </form>";
  }
?>

</body>
</html>
```

例子解释：

1. 首先，检查是否填写了邮件输入框
2. 如果未填写（比如在页面被首次访问时），输出 HTML 表单
3. 如果已填写（在表单被填写后），从表单发送邮件
4. 当点击提交按钮后，重新载入页面，显示邮件发送成功的消息

PHP Mail 参考手册

如需更多有关 PHP mail() 函数的信息，请访问我们的 PHP Mail 参考手册。

PHP 安全的电子邮件

在上一节中的 **PHP e-mail** 脚本中，存在着一个漏洞。

PHP E-mail 注入

首先，请看上一节中的 PHP 代码：

```
<html>
<body>

<?php
if (isset($_REQUEST['email']))
//if "email" is filled out, send email
  {
  //send email
  $email = $_REQUEST['email'] ;
  $subject = $_REQUEST['subject'] ;
  $message = $_REQUEST['message'] ;
  mail("someone@example.com", "Subject: $subject",
  $message, "From: $email" );
  echo "Thank you for using our mail form";
  }
else
//if "email" is not filled out, display the form
  {
  echo "<form method='post' action='mailform.php'>
  Email: <input name='email' type='text' /><br />
  Subject: <input name='subject' type='text' /><br />
  Message:<br />
  <textarea name='message' rows='15' cols='40'>
  </textarea><br />
  <input type='submit' />
  </form>";
  }
?>

</body>
</html>
```

以上代码存在的问题是，未经授权的用户可通过输入表单在邮件头部插入数据。

假如用户在表单中的输入框内加入这些文本，会出现什么情况呢？

```
someone@example.com%0ACc:person2@example.com
%0ABcc:person3@example.com,person3@example.com,
anotherperson4@example.com,person5@example.com
%0ABTo:person6@example.com
```

与往常一样，mail() 函数把上面的文本放入邮件头部，那么现在头部有了额外的 Cc:, Bcc: 以及 To: 字段。当用户点击提交按钮时，这封 e-mail 会被发送到上面所有的地址！

PHP 防止 E-mail 注入

防止 e-mail 注入的最好方法是对输入进行验证。

下面的代码与上一节类似，不过我们已经增加了检测表单中 email 字段的输入验证程序：

```
<html>
<body>
<?php
function spamcheck($field)
  {
  //filter_var() sanitizes the e-mail
  //address using FILTER_SANITIZE_EMAIL
  $field=filter_var($field, FILTER_SANITIZE_EMAIL);

  //filter_var() validates the e-mail
  //address using FILTER_VALIDATE_EMAIL
  if(filter_var($field, FILTER_VALIDATE_EMAIL))
    {
    return TRUE;
    }
  else
    {
    return FALSE;
    }
  }

if (isset($_REQUEST['email']))
  {//if "email" is filled out, proceed

  //check if the email address is invalid
  $mailcheck = spamcheck($_REQUEST['email']);
  if ($mailcheck==FALSE)
    {
    echo "Invalid input";
    }
  else
    {//send email
    $email = $_REQUEST['email'] ;
    $subject = $_REQUEST['subject'] ;
    $message = $_REQUEST['message'] ;
    mail("someone@example.com", "Subject: $subject",
    $message, "From: $email" );
    echo "Thank you for using our mail form";
    }
  }
else
  {//if "email" is not filled out, display the form
  echo "<form method='post' action='mailform.php'>
  Email: <input name='email' type='text' /><br />
  Subject: <input name='subject' type='text' /><br />
  Message:<br />
  <textarea name='message' rows='15' cols='40'>
  </textarea><br />
  <input type='submit' />
  </form>";
  }
?>

</body>
</html>
```

在上面的代码中，我们使用了 PHP 过滤器来对输入进行验证：

- FILTER_SANITIZE_EMAIL 从字符串中删除电子邮件的非法字符
- FILTER_VALIDATE_EMAIL 验证电子邮件地址

您可以在我们的 PHP 过滤器这一节中阅读更多有关过滤器的内容。

PHP 错误处理

在 **PHP** 中，默认的错误处理很简单。一条消息会被发送到浏览器，这条消息带有文件名、行号以及一条描述错误的消息。

PHP 错误处理

在创建脚本和 web 应用程序时，错误处理是一个重要的部分。如果您的代码缺少错误检测编码，那么程序看上去很不专业，也为安全风险敞开了大门。

本教程介绍了 PHP 中一些最为重要的错误检测方法。

我们将为您讲解不同的错误处理方法：

- 简单的 "die()" 语句
- 自定义错误和错误触发器
- 错误报告

基本的错误处理：使用 die() 函数

第一个例子展示了一个打开文本文件的简单脚本：

```
<?php
$file=fopen("welcome.txt","r");
?>
```

如果文件不存在，您会获得类似这样的错误：

```
**Warning**: fopen(welcome.txt) [function.fopen]: failed to open stream:
No such file or directory in **C:\webfolder\test.php** on line **2**
```

为了避免用户获得类似上面的错误消息，我们在访问文件之前检测该文件是否存在：

```
<?php
if(!file_exists("welcome.txt"))
 {
 die("File not found");
 }
else
 {
 $file=fopen("welcome.txt","r");
 }
?>
```

现在，假如文件不存在，您会得到类似这样的错误消息：

```
File not found
```

比起之前的代码，上面的代码更有效，这是由于它采用了一个简单的错误处理机制在错误之后终止了脚本。

不过，简单地终止脚本并不总是恰当的方式。让我们研究一下用于处理错误的备选的 PHP 函数。

创建自定义错误处理器

创建一个自定义的错误处理器非常简单。我们很简单地创建了一个专用函数，可以在 PHP 中发生错误时调用该函数。

该函数必须有能力处理至少两个参数 (error level 和 error message)，但是可以接受最多五个参数（可选的：file, line-number 以及 error context）：

语法

```
error_function(error_level,error_message,
error_file,error_line,error_context)
```

参数	描述
error_level	必需。为用户定义的错误规定错误报告级别。必须是一个值数。 参见下面的表格：错误报告级别。
error_message	必需。为用户定义的错误规定错误消息。
error_file	可选。规定错误在其中发生的文件名。
error_line	可选。规定错误发生的行号。
error_context	可选。规定一个数组，包含了当错误发生时在用的每个变量以及它们的值。

错误报告级别

这些错误报告级别是错误处理程序旨在处理的错误的不同的类型：

值	常量	描述
2	E_WARNING	非致命的 run-time 错误。不暂停脚本执行。
8	E_NOTICE	Run-time 通知。 脚本发现可能有错误发生，但也可能在脚本正常运行时发生。
256	E_USER_ERROR	致命的用户生成的错误。这类似于程序员使用 PHP 函数 trigger_error() 设置的 E_ERROR。
512	E_USER_WARNING	非致命的用户生成的警告。这类似于程序员使用 PHP 函数 trigger_error() 设置的 E_WARNING。
1024	E_USER_NOTICE	用户生成的通知。这类似于程序员使用 PHP 函数 trigger_error() 设置的 E_NOTICE。
4096	E_RECOVERABLE_ERROR	可捕获的致命错误。类似 E_ERROR，但可被用户定义的处理程序捕获。(参见 set_error_handler())
8191	E_ALL	所有错误和警告，除级别 E_STRICT 以外。（在 PHP 6.0，E_STRICT 是 E_ALL 的一部分）

现在，让我们创建一个处理错误的函数：

```
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Ending Script";
 die();
 }
```

上面的代码是一个简单的错误处理函数。当它被触发时，它会取得错误级别和错误消息。然后它会输出错误级别和消息，并终止脚本。

现在，我们已经创建了一个错误处理函数，我们需要确定在何时触发该函数。

Set Error Handler

PHP 的默认错误处理程序是内建的错误处理程序。我们打算把上面的函数改造为脚本运行期间的默认错误处理程序。

可以修改错误处理程序，使其仅应用到某些错误，这样脚本就可以不同的方式来处理不同的错误。不过，在本例中，我们打算针对所有错误来使用我们的自定义错误处理程序：

```
set_error_handler("customError");
```

由于我们希望我们的自定义函数来处理所有错误，set_error_handler() 仅需要一个参数，可以添加第二个参数来规定错误级别。

实例

通过尝试输出不存在的变量，来测试这个错误处理程序：

```
<?php
//error handler function
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr";
 }

//set error handler
set_error_handler("customError");

//trigger error
echo($test);
?>
```

以上代码的输出应该类似这样：

```
Error: [8] Undefined variable: test
```

触发错误

在脚本中用户输入数据的位置，当用户的输入无效时触发错误的很有用的。在 PHP 中，这个任务由 trigger_error() 完成。

例子

在本例中，如果 "test" 变量大于 "1"，就会发生错误：

```
<?php
$test=2;
if ($test>1)
{
trigger_error("Value must be 1 or below");
}
?>
```

以上代码的输出应该类似这样：

```
Notice: Value must be 1 or below
in C:\webfolder\test.php on line 6
```

您可以在脚本中任何位置触发错误，通过添加的第二个参数，您能够规定所触发的错误级别。

可能的错误类型：

- E_USER_ERROR - 致命的用户生成的 run-time 错误。错误无法恢复。脚本执行被中断。
- E_USER_WARNING - 非致命的用户生成的 run-time 警告。脚本执行不被中断。
- E_USER_NOTICE - 默认。用户生成的 run-time 通知。脚本发现了可能的错误，也有可能在脚本运行正常时发生。

例子

在本例中，如果 "test" 变量大于 "1"，则发生 E_USER_WARNING 错误。如果发生了 E_USER_WARNING，我们将使用我们的自定义错误处理程序并结束脚本：

```
<?php
//error handler function
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Ending Script";
 die();
 }

//set error handler
set_error_handler("customError",E_USER_WARNING);

//trigger error
$test=2;
if ($test>1)
 {
 trigger_error("Value must be 1 or below",E_USER_WARNING);
 }
?>
```

以上代码的输出应该类似这样：

```
Error: [512] Value must be 1 or below
Ending Script
```

现在，我们已经学习了如何创建自己的 error，以及如何触发它们，现在我们研究一下错误记录。

错误记录

默认地，根据在 php.ini 中的 error_log 配置，PHP 向服务器的错误记录系统或文件发送错误记录。通过使用 error_log() 函数，您可以向指定的文件或远程目的地发送错误记录。

通过电子邮件向您自己发送错误消息，是一种获得指定错误的通知的好办法。

通过 E-Mail 发送错误消息

在下面的例子中，如果特定的错误发生，我们将发送带有错误消息的电子邮件，并结束脚本：

```
<?php
//error handler function
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Webmaster has been notified";
 error_log("Error: [$errno] $errstr",1,
 "someone@example.com","From: webmaster@example.com");
}

//set error handler
set_error_handler("customError",E_USER_WARNING);

//trigger error
$test=2;
if ($test>1)
 {
 trigger_error("Value must be 1 or below",E_USER_WARNING);
 }
?>
```

以上代码的输出应该类似这样：

```
Error: [512] Value must be 1 or below
Webmaster has been notified
```

接收自以上代码的邮件类似这样：

```
Error: [512] Value must be 1 or below
```

这个方法不适合所有的错误。常规错误应当通过使用默认的 PHP 记录系统在服务器上进行记录。

PHP 异常处理

异常（**Exception**）用于在指定的错误发生时改变脚本的正常流程。

什么是异常？

PHP 5 提供了一种新的面向对象的错误处理方法。

异常处理用于在指定的错误（异常）情况发生时改变脚本的正常流程。这种情况称为异常。

当异常被触发时，通常会发生：

- 当前代码状态被保存
- 代码执行被切换到预定义的异常处理器函数
- 根据情况，处理器也许会从保存的代码状态重新开始执行代码，终止脚本执行，或从代码中另外的位置继续执行脚本

我们将展示不同的错误处理方法：

- 异常的基本使用
- 创建自定义的异常处理器
- 多个异常
- 重新抛出异常
- 设置顶层异常处理器

异常的基本使用

当异常被抛出时，其后的代码不会继续执行，PHP 会尝试查找匹配的 "catch" 代码块。

如果异常没有被捕获，而且又没用使用 set_exception_handler() 作相应的处理的话，那么将发生一个严重的错误（致命错误），并且输出 "Uncaught Exception" （未捕获异常）的错误消息。

让我们尝试抛出一个异常，同时不去捕获它：

```
<?php
//create function with an exception
function checkNum($number)
 {
 if($number>1)
  {
  throw new Exception("Value must be 1 or below");
  }
 return true;
 }

//trigger exception
checkNum(2);
?>
```

上面的代码会获得类似这样的一个错误：

```
Fatal error: Uncaught exception 'Exception'
with message 'Value must be 1 or below' in C:\webfolder\test.php:6
Stack trace: #0 C:\webfolder\test.php(12):
checkNum(28) #1 {main} thrown in C:\webfolder\test.php on line 6
```

Try, throw 和 catch

要避免上面例子出现的错误，我们需要创建适当的代码来处理异常。

正确的处理程序应当包括：

1. Try - 使用异常的函数应该位于 "try" 代码块内。如果没有触发异常，则代码将照常继续执行。但是如果异常被触发，会抛出一个异常。
2. Throw - 这里规定如何触发异常。每一个 "throw" 必须对应至少一个 "catch"
3. Catch - "catch" 代码块会捕获异常，并创建一个包含异常信息的对象

让我们触发一个异常：

```
<?php
//创建可抛出一个异常的函数
function checkNum($number)
 {
 if($number>1)
  {
  throw new Exception("Value must be 1 or below");
  }
 return true;
 }

//在 "try" 代码块中触发异常
try
 {
 checkNum(2);
 //If the exception is thrown, this text will not be shown
 echo 'If you see this, the number is 1 or below';
 }

//捕获异常
catch(Exception $e)
 {
 echo 'Message: ' .$e->getMessage();
 }
?>
```

上面代码将获得类似这样一个错误：

```
Message: Value must be 1 or below
```

例子解释：

上面的代码抛出了一个异常，并捕获了它：

1. 创建 checkNum() 函数。它检测数字是否大于 1。如果是，则抛出一个异常。
2. 在 "try" 代码块中调用 checkNum() 函数。
3. checkNum() 函数中的异常被抛出
4. "catch" 代码块接收到该异常，并创建一个包含异常信息的对象 ($e)。
5. 通过从这个 exception 对象调用 $e->getMessage()，输出来自该异常的错误消息

不过，为了遵循"每个 throw 必须对应一个 catch"的原则，可以设置一个顶层的异常处理器来处理漏掉的错误。

创建一个自定义的 Exception 类

创建自定义的异常处理程序非常简单。我们简单地创建了一个专门的类，当 PHP 中发生异常时，可调用其函数。该类必须是 exception 类的一个扩展。

这个自定义的 exception 类继承了 PHP 的 exception 类的所有属性，您可向其添加自定义的函数。

我们开始创建 exception 类：

```
<?php
class customException extends Exception
 {
 public function errorMessage()
  {
  //error message
  $errorMsg = 'Error on line '.$this->getLine().' in '.$this->getFile()
  .': <b>'.$this->getMessage().'</b> is not a valid E-Mail address';
  return $errorMsg;
  }
 }

$email = "someone@example...com";

try
 {
 //check if
 if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE)
  {
  //throw exception if email is not valid
  throw new customException($email);
  }
 }

catch (customException $e)
 {
 //display custom message
 echo $e->errorMessage();
 }
?>
```

这个新的类是旧的 exception 类的副本，外加 errorMessage() 函数。正因为它是旧类的副本，因此它从旧类继承了属性和方法，我们可以使用 exception 类的方法，比如 getLine() 、getFile() 以及 getMessage()。

例子解释：

上面的代码抛出了一个异常，并通过一个自定义的 exception 类来捕获它：

1. customException() 类是作为旧的 exception 类的一个扩展来创建的。这样它就继承了旧类的所有属性和方法。
2. 创建 errorMessage() 函数。如果 e-mail 地址不合法，则该函数返回一条错误消息
3. 把 $email 变量设置为不合法的 e-mail 地址字符串
4. 执行 "try" 代码块，由于 e-mail 地址不合法，因此抛出一个异常
5. "catch" 代码块捕获异常，并显示错误消息

多个异常

可以为一段脚本使用多个异常，来检测多种情况。

可以使用多个 if..else 代码块，或一个 switch 代码块，或者嵌套多个异常。这些异常能够使用不同的 exception 类，并返回不同的错误消息：

```
<?php
class customException extends Exception
{
public function errorMessage()
{
//error message
$errorMsg = 'Error on line '.$this->getLine().' in '.$this->getFile()
.': <b>'.$this->getMessage().'</b> is not a valid E-Mail address';
return $errorMsg;
}
}

$email = "someone@example.com";

try
 {
 //check if
 if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE)
  {
  //throw exception if email is not valid
  throw new customException($email);
  }
 //check for "example" in mail address
 if(strpos($email, "example") !== FALSE)
  {
  throw new Exception("$email is an example e-mail");
  }
 }

catch (customException $e)
 {
 echo $e->errorMessage();
 }

catch(Exception $e)
 {
 echo $e->getMessage();
 }
?>
```

例子解释：

上面的代码测试了两种条件，如何任何条件不成立，则抛出一个异常：

1. customException() 类是作为旧的 exception 类的一个扩展来创建的。这样它就继承了旧类的所有属性和方法。
2. 创建 errorMessage() 函数。如果 e-mail 地址不合法，则该函数返回一个错误消息。
3. 执行 "try" 代码块，在第一个条件下，不会抛出异常。
4. 由于 e-mail 含有字符串 "example"，第二个条件会触发异常。
5. "catch" 代码块会捕获异常，并显示恰当的错误消息

如果没有捕获 customException，紧紧捕获了 base exception，则在那里处理异常。

重新抛出异常

有时，当异常被抛出时，您也许希望以不同于标准的方式对它进行处理。可以在一个 "catch" 代码块中再次抛出异常。

脚本应该对用户隐藏系统错误。对程序员来说，系统错误也许很重要，但是用户对它们并不感兴趣。为了让用户更容易使用，您可以再次抛出带有对用户比较友好的消息的异常：

```
<?php
class customException extends Exception
 {
 public function errorMessage()
  {
  //error message
  $errorMsg = $this->getMessage().' is not a valid E-Mail address.';
  return $errorMsg;
  }
 }

$email = "someone@example.com";

try
 {
 try
  {
  //check for "example" in mail address
  if(strpos($email, "example") !== FALSE)
   {
   //throw exception if email is not valid
   throw new Exception($email);
   }
  }
 catch(Exception $e)
  {
  //re-throw exception
  throw new customException($email);
  }
 }

catch (customException $e)
 {
 //display custom message
 echo $e->errorMessage();
 }
?>
```

例子解释：

上面的代码检测在邮件地址中是否含有字符串 "example"。如果有，则再次抛出异常：

1. customException() 类是作为旧的 exception 类的一个扩展来创建的。这样它就继承了旧类的所有属性和方法。
2. 创建 errorMessage() 函数。如果 e-mail 地址不合法，则该函数返回一个错误消息。
3. 把 $email 变量设置为一个有效的邮件地址，但含有字符串 "example"。
4. "try" 代码块包含另一个 "try" 代码块，这样就可以再次抛出异常。
5. 由于 e-mail 包含字符串 "example"，因此触发异常。
6. "catch" 捕获到该异常，并重新抛出 "customException"。
7. 捕获到 "customException"，并显示一条错误消息。

如果在其目前的 "try" 代码块中异常没有被捕获，则它将在更高层级上查找 catch 代码块。

设置顶层异常处理器 （Top Level Exception Handler）

set_exception_handler() 函数可设置处理所有未捕获异常的用户定义函数。

```
<?php
function myException($exception)
{
echo "<b>Exception:</b> " , $exception->getMessage();
}

set_exception_handler('myException');

throw new Exception('Uncaught Exception occurred');
?>
```

以上代码的输出应该类似这样：

```
Exception: Uncaught Exception occurred
```

在上面的代码中，不存在 "catch" 代码块，而是触发顶层的异常处理程序。应该使用此函数来捕获所有未被捕获的异常。

异常的规则

- 需要进行异常处理的代码应该放入 try 代码块内，以便捕获潜在的异常。
- 每个 try 或 throw 代码块必须至少拥有一个对应的 catch 代码块。
- 使用多个 catch 代码块可以捕获不同种类的异常。
- 可以在 try 代码块内的 catch 代码块中再次抛出（re-thrown）异常。

简而言之：如果抛出了异常，就必须捕获它。

PHP 过滤器（Filter）

- MySQL 简介

PHP 过滤器用于验证和过滤来自非安全来源的数据，比如用户的输入。

什么是 PHP 过滤器？

PHP 过滤器用于验证和过滤来自非安全来源的数据。

验证和过滤用户输入或自定义数据是任何 Web 应用程序的重要组成部分。

设计 PHP 的过滤器扩展的目的是使数据过滤更轻松快捷。

为什么使用过滤器？

几乎所有 web 应用程序都依赖外部的输入。这些数据通常来自用户或其他应用程序（比如 web 服务）。通过使用过滤器，您能够确保应有程序获得正确的输入类型。

您应该始终对外部数据进行过滤！

输入过滤是最重要的应用程序安全课题之一。

什么是外部数据？

- 来自表单的输入数据
- Cookies
- 服务器变量
- 数据库查询结果

函数和过滤器

如需过滤变量，请使用下面的过滤器函数之一：

- filter_var() - 通过一个指定的过滤器来过滤单一的变量
- filter_var_array() - 通过相同的或不同的过滤器来过滤多个变量
- filter_input - 获取一个输入变量，并对它进行过滤
- filter_input_array - 获取多个输入变量，并通过相同的或不同的过滤器对它们进行过滤

在下面的例子中，我们用 filter_var() 函数验证了一个整数：

```
<?php
$int = 123;

if(!filter_var($int, FILTER_VALIDATE_INT))
 {
 echo("Integer is not valid");
 }
else
 {
 echo("Integer is valid");
 }
?>
```

上面的代码使用了 "FILTER_VALIDATE_INT" 过滤器来过滤变量。由于这个整数是合法的，因此代码的输出是："Integer is valid"。

假如我们尝试使用一个非整数的变量，则输出是："Integer is not valid"。

如需完整的函数和过滤器列表，请访问我们的 PHP Filter 参考手册。

Validating 和 Sanitizing

有两种过滤器：

Validating 过滤器：

- 用于验证用户输入
- 严格的格式规则（比如 URL 或 E-Mail 验证）
- 如果成功则返回预期的类型，如果失败则返回 FALSE

Sanitizing 过滤器：

- 用于允许或禁止字符串中指定的字符
- 无数据格式规则
- 始终返回字符串

选项和标志

选项和标志用于向指定的过滤器添加额外的过滤选项。

不同的过滤器有不同的选项和标志。

在下面的例子中，我们用 filter_var() 和 "min_range" 以及 "max_range" 选项验证了一个整数：

```
<?php
$var=300;
 $int_options = array(
"options"=>array
 (
 "min_range"=>0,
 "max_range"=>256
 )
);
if(!filter_var($var, FILTER_VALIDATE_INT, $int_options))
 {
 echo("Integer is not valid");
 }
else
 {
 echo("Integer is valid");
 }
?>
```

就像上面的代码一样，选项必须放入一个名为 "options" 的相关数组中。如果使用标志，则不需在数组内。

由于整数是 "300"，它不在指定的范围内，以上代码的输出将是 "Integer is not valid"。

如需完整的函数及过滤器列表，请访问 W3School 提供的 PHP Filter 参考手册。您可以看到每个过滤器的可用选项和标志。

验证输入

让我们试着验证来自表单的输入。

我们需要作的第一件事情是确认是否存在我们正在查找的输入数据。

然后我们用 filter_input() 函数过滤输入的数据。

在下面的例子中，输入变量 "email" 被传到 PHP 页面：

```
<?php
if(!filter_has_var(INPUT_GET, "email"))
 {
 echo("Input type does not exist");
 }
else
 {
 if (!filter_input(INPUT_GET, "email", FILTER_VALIDATE_EMAIL))
  {
  echo "E-Mail is not valid";
  }
 else
  {
  echo "E-Mail is valid";
  }
 }
?>
```

例子解释：

上面的例子有一个通过 "GET" 方法传送的输入变量 (email)：

1. 检测是否存在 "GET" 类型的 "email" 输入变量
2. 如果存在输入变量，检测它是否是有效的邮件地址

净化输入

让我们试着清理一下从表单传来的 URL。

首先，我们要确认是否存在我们正在查找的输入数据。

然后，我们用 filter_input() 函数来净化输入数据。

在下面的例子中，输入变量 "url" 被传到 PHP 页面：

```
<?php
if(!filter_has_var(INPUT_POST, "url"))
 {
 echo("Input type does not exist");
 }
else
 {
 $url = filter_input(INPUT_POST, "url", FILTER_SANITIZE_URL);
 }
?>
```

例子解释：

上面的例子有一个通过 "POST" 方法传送的输入变量 (url)：

1. 检测是否存在 "POST" 类型的 "url" 输入变量
2. 如果存在此输入变量，对其进行净化（删除非法字符），并将其存储在 $url 变量中

假如输入变量类似这样："http://www.W3非o法ol.com.c字符n/"，则净化后的 $url 变量应该是这样的：

```
http://www.W3School.com.cn/
```

过滤多个输入

表单通常由多个输入字段组成。为了避免对 filter_var 或 filter_input 重复调用，我们可以使用 filter_var_array 或 the filter_input_array 函数。

在本例中，我们使用 filter_input_array() 函数来过滤三个 GET 变量。接收到的 GET 变量是一个名字、一个年龄以及一个邮件地址：

```
<?php
$filters = array
 (
 "name" => array
  (
  "filter"=>FILTER_SANITIZE_STRING
  ),
 "age" => array
  (
  "filter"=>FILTER_VALIDATE_INT,
  "options"=>array
   (
   "min_range"=>1,
   "max_range"=>120
   )
  ),
 "email"=> FILTER_VALIDATE_EMAIL,
 );

$result = filter_input_array(INPUT_GET, $filters);

if (!$result["age"])
 {
 echo("Age must be a number between 1 and 120.<br />");
 }
elseif(!$result["email"])
 {
 echo("E-Mail is not valid.<br />");
 }
else
 {
 echo("User input is valid");
 }
?>
```

例子解释：

上面的例子有三个通过 "GET" 方法传送的输入变量 (name, age and email)

1. 设置一个数组，其中包含了输入变量的名称，以及用于指定的输入变量的过滤器
2. 调用 filter_input_array 函数，参数包括 GET 输入变量及刚才设置的数组
3. 检测 $result 变量中的 "age" 和 "email" 变量是否有非法的输入。（如果存在非法输入，）

filter_input_array() 函数的第二个参数可以是数组或单一过滤器的 ID。

如果该参数是单一过滤器的 ID，那么这个指定的过滤器会过滤输入数组中所有的值。

如果该参数是一个数组，那么此数组必须遵循下面的规则：

- 必须是一个关联数组，其中包含的输入变量是数组的键（比如 "age" 输入变量）
- 此数组的值必须是过滤器的 ID ，或者是规定了过滤器、标志以及选项的数组

使用 **Filter Callback**

通过使用 FILTER_CALLBACK 过滤器，可以调用自定义的函数，把它作为一个过滤器来使用。这样，我们就拥有了数据过滤的完全控制权。

您可以创建自己的自定义函数，也可以使用已有的 PHP 函数。

规定您准备用到过滤器函数的方法，与规定选项的方法相同。

在下面的例子中，我们使用了一个自定义的函数把所有 "_" 转换为空格：

```
<?php
function convertSpace($string)
{
return str_replace("_", " ", $string);
}

$string = "Peter_is_a_great_guy!";

echo filter_var($string, FILTER_CALLBACK, array("options"=>"convertSpace"));
?>
```

以上代码的结果是这样的：

```
Peter is a great guy!
```

例子解释：

上面的例子把所有 "_" 转换成空格：

1. 创建一个把 "_" 替换为空格的函数
2. 调用 filter_var() 函数，它的参数是 FILTER_CALLBACK 过滤器以及包含我们的函数的数组

- MySQL 简介

PHP JSON

本章节我们将为大家介绍如何使用 PHP 语言来编码和解码 JSON 对象。

环境配置

在 php5.2.0 及以上版本已经内置 JSON 扩展。

JSON 函数

函数	描述
json_encode	对变量进行 JSON 编码
json_decode	对 JSON 格式的字符串进行解码，转换为 PHP 变量
json_last_error	返回最后发生的错误

json_encode

PHP json_encode() 用于对变量进行 JSON 编码，该函数如果执行成功返回 JSON 数据，否则返回 FALSE 。

语法

```
string json_encode ( $value [, $options = 0 ] )
```

参数

- **value**: 要编码的值。该函数只对 UTF-8 编码的数据有效。
- **options**:由以下常量组成的二进制掩码：JSON_HEX_QUOT, JSON_HEX_TAG, JSON_HEX_AMP, JSON_HEX_APOS, JSON_NUMERIC_CHECK,JSON_PRETTY_PRINT, JSON_UNESCAPED_SLASHES, JSON_FORCE_OBJECT

实例

以下实例演示了如何将 PHP 数组转换为 JSON 格式数据：

```
<?php
   $arr = array('a' => 1, 'b' => 2, 'c' => 3, 'd' => 4, 'e' => 5);
   echo json_encode($arr);
?>
```

以上代码执行结果为：

```
{"a":1,"b":2,"c":3,"d":4,"e":5}
```

以下实例演示了如何将 PHP 对象转换为 JSON 格式数据：

```
?php
   class Emp {
       public $name = "";
       public $hobbies  = "";
       public $birthdate = "";
   }
   $e = new Emp();
   $e->name = "sachin";
   $e->hobbies  = "sports";
   $e->birthdate = date('m/d/Y h:i:s a', "8/5/1974 12:20:03 p");
   $e->birthdate = date('m/d/Y h:i:s a', strtotime("8/5/1974 12:20:03"));

   echo json_encode($e);
?>
```

以上代码执行结果为：

```
{"name":"sachin","hobbies":"sports","birthdate":"08\/05\/1974 12:20:03 pm"}
```

json_decode

PHP json_decode() 函数用于对 JSON 格式的字符串进行解码，并转换为 PHP 变量。

语法

```
mixed json_decode ($json [,$assoc = false [, $depth = 512 [, $options = 0 ]]])
```

参数

- **json_string**: 待解码的 JSON 字符串，必须是 UTF-8 编码数据
- **assoc**: 当该参数为 TRUE 时，将返回数组，FALSE 时返回对象。
- **depth**: 整数类型的参数，它指定递归深度
- **options**: 二进制掩码，目前只支持 JSON_BIGINT_AS_STRING 。

实例

以下实例演示了如何解码 JSON 数据：

```
<?php
   $json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

   var_dump(json_decode($json));
   var_dump(json_decode($json, true));
?>
```

以上代码执行结果为：

```
object(stdClass)#1 (5) {
    ["a"] => int(1)
    ["b"] => int(2)
    ["c"] => int(3)
    ["d"] => int(4)
    ["e"] => int(5)
}

array(5) {
    ["a"] => int(1)
    ["b"] => int(2)
    ["c"] => int(3)
    ["d"] => int(4)
    ["e"] => int(5)
}
```

PHP 数据库

PHP MySQL 简介

通过 PHP，您可以连接和操作数据库。

MySQL 是跟 PHP 配套使用的最流行的开源数据库系统。

如果想学习更多 MySQL 知识可以查看本站 MySQL 教程。

MySQL 是什么？

- MySQL 是一种在 Web 上使用的数据库系统。
- MySQL 是一种在服务器上运行的数据库系统。
- MySQL 不管在小型还是大型应用程序中，都是理想的选择。
- MySQL 是非常快速，可靠，且易于使用的。
- MySQL 支持标准的 SQL。
- MySQL 在一些平台上编译。
- MySQL 是免费下载使用的。
- MySQL 是由 Oracle 公司开发、发布和支持的。
- MySQL 是以公司创始人 Monty Widenius's daughter: My 命名的。

MySQL 中的数据存储在表中。表格是一个相关数据的集合，它包含了列和行。

在分类存储信息时，数据库非常有用。一个公司的数据库可能拥有以下表：

- Employees
- Products
- Customers
- Orders

PHP + MySQL

- PHP 与 MySQL 结合是跨平台的。（您可以在 Windows 上开发，在 Unix 平台上应用。）

查询

查询是一种询问或请求。

通过 MySQL，我们可以向数据库查询具体的信息，并得到返回的记录集。

请看下面的查询（使用标准 SQL）：

```
SELECT LastName FROM Employees
```

上面的查询选取了 "Employees" 表中 "LastName" 列的所有数据。

如需学习更多关于 SQL 的知识，请访问我们的 SQL 教程。

下载 MySQL 数据库

如果您的 PHP 服务器没有 MySQL 数据库，可以在此免费下载 MySQL：http://www.mysql.com。

关于 MySQL 数据库的事实

关于 MySQL 的一点很棒的特性是，可以对它进行缩减，来支持嵌入的数据库应用程序。也许正因为如此，许多人认为 MySQL 仅仅能处理中小型的系统。

事实上，对于那些支持巨大数据和访问量的网站（比如 Friendster、Yahoo、Google），MySQL 是事实上的标准数据库。

这个地址提供了使用 MySQL 的公司的概览： http://www.mysql.com/customers/。

PHP 连接 MySQL

PHP 5 及以上版本建议使用以下方式连接 MySQL :

- **MySQLi extension** ("i" 意为 improved)
- **PDO (PHP Data Objects)**

在 PHP 早起版本中我们使用 MySQL 扩展。但该扩展在 2012 年开始不建议使用。

我是该用 MySQLi ， 还是 PDO?

如果你需要一个简短的回答，即 "你习惯哪个就用哪个"。

MySQLi 和 PDO 有它们自己的优势 :

PDO 应用在 12 种不同数据库中， MySQLi 只针对 MySQL 数据库。

所以，如果你的项目需要在多种数据库中切换，建议使用 PDO ，这样你只需要修改连接字符串和部门查询语句即可。 使用 MySQLi, 如果不同数据库，你需要重新所有代码，包括查询。

两者都是面向对象, 但 MySQLi 还提供了 API 接口。

两者都支持预处理语句。 预处理语句可以防止 SQL 注入，对于 web 项目的安全性是非常重要的。

MySQLi 和 PDO 连接 MySQL 实例

在本章节及接下来的章节中，我们会使用以下三种方式来演示 PHP 操作 MySQL:

- MySQLi (面向对象)
- MySQLi (面向过程)
- PDO

MySQLi Installation

Linux 和 Windows: 在 php5 mysql 包安装时 MySQLi 扩展多事情况下是自动安装的。

安装详细信息，请查看： http://php.net/manual/en/mysqli.installation.php

PDO 安装

For 安装详细信息，请查看： http://php.net/manual/en/pdo.installation.php

连接 MySQL

在我们访问 MySQL 数据库前，我们需要先连接到数据库服务器：

实例 (MySQLi - 面向对象)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";

// 创建连接
$conn = new mysqli($servername, $username, $password);

// 检测连接
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
echo "Connected successfully";
?>
```

> 注意在以上面向对象的实例中 $connect_error 是在 PHP 5.2.9 和 5.3.0 中添加的。如果你需要兼容更早版本 请使用以下代码替换:

```
// 检测连接
if (mysqli_connect_error()) {
    die("Database connection failed: " . mysqli_connect_error());
}
```

实例 (MySQLi - 面向过程)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";

// 创建连接
$conn = mysqli_connect($servername, $username, $password);

// 检测连接
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}
echo "Connected successfully";
?>
```

实例 (PDO)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";

try {
    $conn = new PDO("mysql:host=$servername;dbname=myDB", $username, $password);
    echo "Connected successfully";
    }
catch(PDOException $e)
    {
    echo $e->getMessage();
    }
?>
```

> 注意在以上 PDO 实例中我们已经指定了数据库 (myDB)。PDO 在连接过程需要设置数据库名。如果没有指定，则会抛出异常。

关闭连接

连接在脚本执行完后会自动关闭。你也可以使用以下代码来关闭连接：

实例 (MySQLi - 面向对象)

```
$conn->close();
```

实例 (MySQLi - 面向过程)

```
mysqli_close($conn);
```

实例 (PDO)

```
$conn = null;
```

PHP MySQL 创建数据库

数据库存有一个或多个表。

你需要 CREATE 权限来创建或删除 MySQL 数据库。

使用 MySQLi 和 PDO 创建 MySQL 数据库

CREATE DATABASE 语句用于在 MySQL 中创建数据库。

在下面的实例中，创建了一个名为 "myDB" 的数据库：

实例 (MySQLi - 面向对象)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";

// 创建连接
$conn = new mysqli($servername, $username, $password);
// 检测连接
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

// Create database
$sql = "CREATE DATABASE myDB";
if ($conn->query($sql) === TRUE) {
    echo "Database created successfully";
} else {
    echo "Error creating database: " . $conn->error;
}

$conn->close();
?>
```

> 注意： 当你创建一个新的数据库时，你必须为 mysqli 对象指定三个参数 (servername, username 和 password)。
>
> **Tip:** 如果你使用其他端口（默认为3306），为数据库参数添加空字符串，如: new mysqli("localhost", "username", "password", "", port)

实例 (MySQLi Procedural)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";

// 创建连接
$conn = mysqli_connect($servername, $username, $password);
// 检测连接
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}

// Create database
$sql = "CREATE DATABASE myDB";
if (mysqli_query($conn, $sql)) {
    echo "Database created successfully";
} else {
    echo "Error creating database: " . mysqli_error($conn);
}

mysqli_close($conn);
?>
```

注意： 以下使用 PDO 实例创建数据库 "myDBPDO":

实例 (PDO)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";

try {
    $conn = new PDO("mysql:host=$servername;dbname=myDB", $username, $password);
    // 设置 PDO 错误模式为异常
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $sql = "CREATE DATABASE myDBPDO";
    // 使用 exec() ，因为没有结果返回
    $conn->exec($sql);
    echo "Database created successfully<br>";
    }
catch(PDOException $e)
    {
    echo $sql . "<br>" . $e->getMessage();
    }

$conn = null;
?>
```

提示： 使用 PDO 的最大好处是在数据库查询过程出现问题时可以使用异常类来 处理问题。如果 try{ } 代码块出现异常，脚本会停止执行并会跳到第一个 catch(){ } 代码块执行代码。 在以上捕获的代码块中我们输出了 SQL 语句并生成错误信息。

PHP 创建 MySQL 表

一个数据表有一个唯一名称，并有行和列组成。

使用 MySQLi 和 PDO 创建 MySQL 表

CREATE TABLE 语句用于创建 MySQL 表。

我们将创建一个名为 "MyGuests" 的表，有 5 个列： "id", "firstname", "lastname", "email" 和 "reg_date":

```
CREATE TABLE MyGuests (
id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
firstname VARCHAR(30) NOT NULL,
lastname VARCHAR(30) NOT NULL,
email VARCHAR(50),
reg_date TIMESTAMP
)
```

上表中的注意事项:

数据类型指定列可以存储什么类型的数据。完整的数据类型请参考我们的 数据类型参考手册。

在设置了数据类型后，你可以为没个列指定其他选项的属性：

- NOT NULL - 没一行都必须含有值（不能为空），null 值是不允许的。
- DEFAULT value - 设置默认值
- UNSIGNED - 使用无符号数值类型，0 及正数
- AUTO INCREMENT - 设置 MySQL 字段的值在新增记录时每次自动增长 1
- PRIMARY KEY - 设置数据表中每条记录的唯一标识。 通常列的 PRIMARY KEY 设置为 ID 数值，与 AUTO_INCREMENT 一起使用。

每个表都应该有一个主键(本列为 "id" 列)，主键必须包含唯一的值。

以下实例展示了如何在 PHP 中创建表：

实例 (MySQLi - 面向对象)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建连接
$conn = new mysqli($servername, $username, $password, $dbname);
// 检测连接
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

// sql to create table
$sql = "CREATE TABLE MyGuests (
id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
firstname VARCHAR(30) NOT NULL,
lastname VARCHAR(30) NOT NULL,
email VARCHAR(50),
reg_date TIMESTAMP
)";

if ($conn->query($sql) === TRUE) {
    echo "Table MyGuests created successfully";
} else {
    echo "Error creating table: " . $conn->error;
}

$conn->close();
?>
```

实例 (MySQLi - 面向过程)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建连接
$conn = mysqli_connect($servername, $username, $password, $dbname);
// 检测连接
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}

// sql to create table
$sql = "CREATE TABLE MyGuests (
id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
firstname VARCHAR(30) NOT NULL,
lastname VARCHAR(30) NOT NULL,
email VARCHAR(50),
reg_date TIMESTAMP
)";

if (mysqli_query($conn, $sql)) {
    echo "Table MyGuests created successfully";
} else {
    echo "Error creating table: " . mysqli_error($conn);
}

mysqli_close($conn);
?>
```

实例 (PDO)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDBPDO";

try {
    $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);
    // set the PDO error mode to exception
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    // sql to create table
    $sql = "CREATE TABLE MyGuests (
    id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    firstname VARCHAR(30) NOT NULL,
    lastname VARCHAR(30) NOT NULL,
    email VARCHAR(50),
    reg_date TIMESTAMP
    )";

    // use exec() because no results are returned
    $conn->exec($sql);
    echo "Table MyGuests created successfully";
    }
catch(PDOException $e)
    {
    echo $sql . "<br>" . $e->getMessage();
    }

$conn = null;
?>
```

PHP MySQL 插入数据

使用 MySQLi 和 PDO 向 MySQL 插入数据

在创建完数据库和表后，我们可以向表中添加数据。

以下为一些语法规则：

- PHP 中 SQL 查询语句必须使用引号
- 在 SQL 查询语句中的字符串值必须加引号
- 数值的值不需要引号
- NULL 值不需要引号

INSERT INTO 语句通常用于向 MySQL 表添加新的记录：

```
INSERT INTO table_name (column1, column2, column3,...)
VALUES (value1, value2, value3,...)
```

学习更多关于 SQL 知识，请查看我们的 SQL 教程。

在前面的几个章节中我们已经创建了表 "MyGuests"，表字段有: "id", "firstname", "lastname", "email" 和 "reg_date"。 现在，让我们开始向表填充数据。

> 注意： 如果列设置 AUTO_INCREMENT (如 "id" 列) 或 TIMESTAMP (如 "reg_date" 列),， 我们就不需要在 SQL 查询语句中指定值； MySQL 会自动为该列添加值。

以下实例向 "MyGuests" 表添加了新的记录:

实例 (MySQLi - 面向对象)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建连接
$conn = new mysqli($servername, $username, $password, $dbname);
// 检测连接
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$sql = "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('John', 'Doe', 'john@example.com')";

if ($conn->query($sql) === TRUE) {
    echo "New record created successfully";
} else {
    echo "Error: " . $sql . "<br>" . $conn->error;
}

$conn->close();
?>
```

实例 (MySQLi - 面向过程)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建连接
$conn = mysqli_connect($servername, $username, $password, $dbname);
// 检测连接
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}

$sql = "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('John', 'Doe', 'john@example.com')";

if (mysqli_query($conn, $sql)) {
    echo "New record created successfully";
} else {
    echo "Error: " . $sql . "<br>" . mysqli_error($conn);
}

mysqli_close($conn);
?>
```

实例 (PDO)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDBPDO";

try {
    $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);
    // set the PDO error mode to exception
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $sql = "INSERT INTO MyGuests (firstname, lastname, email)
    VALUES ('John', 'Doe', 'john@example.com')";
    // use exec() because no results are returned
    $conn->exec($sql);
    echo "New record created successfully";
    }
catch(PDOException $e)
    {
    echo $sql . "<br>" . $e->getMessage();
    }

$conn = null;
?>
```

PHP MySQL 插入多条数据

使用 MySQLi 和 PDO 向 MySQL 插入多条数据

mysqli_multi_query() 函数可用来执行多条SQL语句。

以下实例向 "MyGuests" 表添加了三条新的记录:

实例 (MySQLi - 面向对象)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建链接
$conn = new mysqli($servername, $username, $password, $dbname);
// 检查链接
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$sql = "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('John', 'Doe', 'john@example.com');";
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('Mary', 'Moe', 'mary@example.com');";
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('Julie', 'Dooley', 'julie@example.com')";

if ($conn->multi_query($sql) === TRUE) {
    echo "New records created successfully";
} else {
    echo "Error: " . $sql . "<br>" . $conn->error;
}

$conn->close();
?>
```

> 请注意，每个SQL语句必须用分号隔开。

实例 (MySQLi - 面向过程)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建链接
$conn = mysqli_connect($servername, $username, $password, $dbname);
// 检查链接
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}

$sql = "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('John', 'Doe', 'john@example.com');";
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('Mary', 'Moe', 'mary@example.com');";
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('Julie', 'Dooley', 'julie@example.com')";

if (mysqli_multi_query($conn, $sql)) {
    echo "New records created successfully";
} else {
    echo "Error: " . $sql . "<br>" . mysqli_error($conn);
}

mysqli_close($conn);
?>
```

实例 (PDO)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDBPDO";

try {
    $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);
    // set the PDO error mode to exception
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    // 开始事务
    $conn->beginTransaction();
    // SQL 语句
    $conn->exec("INSERT INTO MyGuests (firstname, lastname, email)
    VALUES ('John', 'Doe', 'john@example.com')");
    $conn->exec("INSERT INTO MyGuests (firstname, lastname, email)
    VALUES ('Mary', 'Moe', 'mary@example.com')");
    $conn->exec("INSERT INTO MyGuests (firstname, lastname, email)
    VALUES ('Julie', 'Dooley', 'julie@example.com')");

    // commit the transaction
    $conn->commit();
    echo "New records created successfully";
    }
catch(PDOException $e)
    {
    // roll back the transaction if something failed
    $conn->rollback();
    echo $sql . "<br>" . $e->getMessage();
    }

$conn = null;
?>
```

使用预处理语句

mysqli 扩展提供了第二种方式用于插入语句。

我们可以预处理语句及绑定参数。

mysql 扩展可以不带数据发送语句或查询到mysql数据库。 你可以向列关联或 "绑定" 变量。

Example (MySQLi 使用预处理语句)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);
// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
} else {
    $sql = "INSERT INTO MyGuests VALUES(?, ?, ?)";

    // 为 mysqli_stmt_prepare() 初始化 statement 对象
    $stmt = mysqli_stmt_init($conn);

    //预处理语句
    if (mysqli_stmt_prepare($stmt, $sql)) {
        // 绑定参数
        mysqli_stmt_bind_param($stmt, 'sss', $firstname, $lastname, $email);

        // 设置参数并执行
        $firstname = 'John';
        $lastname = 'Doe';
        $email = 'john@example.com';
        mysqli_stmt_execute($stmt);

        $firstname = 'Mary';
        $lastname = 'Moe';
        $email = 'mary@example.com';
        mysqli_stmt_execute($stmt);

        $firstname = 'Julie';
        $lastname = 'Dooley';
        $email = 'julie@example.com';
        mysqli_stmt_execute($stmt);
    }
}
?>
```

我们可以看到以上实例中使用模块化来处理问题。我们可以通过创建代码块实现更简单的读取和管理。

注意参数的绑定。让我们看下 mysqli_stmt_bind_param() 中的代码：

```
mysqli_stmt_bind_param($stmt, 'sss', $firstname, $lastname, $email);
```

该函数绑定参数查询并将参数传递给数据库。第二个参数是 "sss" 。以下列表展示了参数的类型。 s 字符告诉 mysql 参数是字符串。

This argument may be one of four types:

- i - integer
- d - double
- s - string
- b - BLOB

每个参数必须指定类型，来保证数据的安全性。通过类型的判断可以减少SQL注入漏洞带来的风险。

PHP MySQL 预处理语句

预处理语句对于防止 MySQL 注入是非常有用的。

预处理语句及绑定参数

预处理语句用于执行多个相同的 SQL 语句，并且执行效率更高。

预处理语句的工作原理如下：

1. 预处理：创建 SQL 语句模板并发送到数据库。预留的值使用参数 "?" 标记 。例如：INSERT INTO MyGuests VALUES(?, ?, ?)
2. 数据库解析，编译，对SQL语句模板执行查询优化，并存储结果不输出
3. 执行：最后，将应用绑定的值传递给参数（"?" 标记），数据库执行语句。应用可以多次执行语句，如果参数的值不一样。

相比于直接执行SQL语句，预处理语句有两个主要优点：

- 预处理语句大大减少了分析时间，只做了一次查询（虽然语句多次执行）
- 绑定参数减少了服务器带宽，你只需要发送查询的参数，而不是整个语句
- 预处理语句针对SQL注入是非常有用的，因为 参数值发送后使用不同的协议，保证了数据的合法性。

MySQLi 预处理语句

以下实例在 MySQLi 中使用了预处理语句，并绑定了相应的参数:

实例 (MySQLi 使用预处理语句)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建连接
$conn = new mysqli($servername, $username, $password, $dbname);

// 检测连接
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

// prepare and bind
$stmt = $conn->prepare("INSERT INTO MyGuests VALUES(?, ?, ?)");
$stmt->bind_param("sss", $firstname, $lastname, $email);

// 设置参数并执行
$firstname = "John";
$lastname = "Doe";
$email = "john@example.com";
$stmt->execute();

$firstname = "Mary";
$lastname = "Moe";
$email = "mary@example.com";
$stmt->execute();

$firstname = "Julie";
$lastname = "Dooley";
$email = "julie@example.com";
$stmt->execute();

echo "New records created successfully";

$stmt->close();
$conn->close();
?>
```

解析以下实例的每行代码:

```
"INSERT INTO MyGuests VALUES(?, ?, ?)"
```

在 SQL 语句中，我们使用了问号 (?)，在此我们可以将问号替换为整型，字符串，双精度浮点型和布尔值。

接下来，让我们来看下 bind_param() 函数：

```
$stmt->bind_param("sss", $firstname, $lastname, $email);
```

该函数绑定了 SQL 的参数，且告诉数据库参数的值。 "sss" 参数列处理其余参数的数据类型。s 字符告诉数据库该参数为字符串。

参数有以下四种类型:

- i - integer（整型）
- d - double（双精度浮点型）

- s - string（字符串）
- b - BLOB（布尔值）

每个参数都需要指定类型。

通过告诉数据库参数的数据类型，可以降低 SQL 注入的风险。

> 注意： 如果你想插入其他数据（用户输入），对数据的验证是非常重要的。

PDO 中的预处理语句

以下实例我们在 PDO 中使用了预处理语句并绑定参数:

实例 (PDO 使用预处理语句)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDBPDO";

try {
    $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);
    // 设置 PDO 错误模式为异常
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    // 预处理 SQL 并绑定参数
    $stmt = $conn->prepare("INSERT INTO MyGuests (firstname, lastname, email)
    VALUES (:firstname, :lastname, :email)");
    $stmt->bindParam(':firstname', $firstname);
    $stmt->bindParam(':lastname', $lastname);
    $stmt->bindParam(':email', $email);

    // 插入行
    $firstname = "John";
    $lastname = "Doe";
    $email = "john@example.com";
    $stmt->execute();

    // 插入其他行
    $firstname = "Mary";
    $lastname = "Moe";
    $email = "mary@example.com";
    $stmt->execute();

    // 插入其他行
    $firstname = "Julie";
    $lastname = "Dooley";
    $email = "julie@example.com";
    $stmt->execute();

    echo "New records created successfully";
    }
catch(PDOException $e)
    {
    echo $sql . "<br>" . $e->getMessage();
    }
$conn = null;
?>
```

PHP MySQL 读取数据

从 MySQL 数据库读取数据

SELECT 语句用于从数据表中读取数据:

```
SELECT column_name(s) FROM table_name
```

如需学习更多关于 SQL 的知识，请访问我们的 SQL 教程。

以下实例中我们从表 MyGuests 读取了 id, firstname 和 lastname 列的数据并显示在页面上：

实例 (MySQLi - 面向对象)

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// 创建连接
$conn = new mysqli($servername, $username, $password, $dbname);
// 检测连接
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$sql = "SELECT id, firstname, lastname FROM MyGuests";
$result = $conn->query($sql);

if ($result->num_rows > 0) {
    // 输出每行数据
    while($row = $result->fetch_assoc()) {
        echo "<br> id: ". $row["id"]. " - Name: ". $row["firstname"]. " " . $row["lastnam
    }
} else {
    echo "0 results";
}
$conn->close();
?>
```

以下实例读取了 MyGuests 表的所有记录并显示在 HTML 表格中：

实例 (PDO)

```
<?php
echo "<table style='border: solid 1px black;'>";
echo "<tr><th>Id</th><th>Firstname</th><th>Lastname</th><th>Email</th><th>Reg date</th></

class TableRows extends RecursiveIteratorIterator {
    function __construct($it) {
        parent::__construct($it, self::LEAVES_ONLY);
    }

    function current() {
        return "<td style='width: 150px; border: 1px solid black;'>" . parent::current().
    }

    function beginChildren() {
        echo "<tr>";
    }

    function endChildren() {
        echo "</tr>" . "\n";
    }
}

$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDBPDO";

try {
    $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $stmt = $conn->prepare("SELECT * FROM MyGuests");
    $stmt->execute();

    // 设置结果集为关联数组
    $result = $stmt->setFetchMode(PDO::FETCH_ASSOC);

    foreach(new TableRows(new RecursiveArrayIterator($stmt->fetchAll())) as $k=>$v) {
        echo $v;
    }
    $dsn = null;
    }
catch(PDOException $e)
    {
    echo "Error: " . $e->getMessage();
    }
$conn = null;
echo "</table>";
?>
```

PHP MySQL Where 子句

WHERE 子句用于过滤记录。

WHERE 子句

WHERE 子句用于提取满足指定标准的的记录。

语法

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator value
```

如需学习更多关于 SQL 的知识，请访问我们的 SQL 教程。

为了让 PHP 执行上面的语句，我们必须使用 mysqli_query() 函数。该函数用于向 MySQL 连接发送查询或命令。

实例

下面的实例将从 "Persons" 表中选取所有 FirstName='Peter' 的行：

```
<?php
$con=mysqli_connect("example.com","peter","abc123","my_db");
// Check connection
if (mysqli_connect_errno())
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$result = mysqli_query($con,"SELECT * FROM Persons
WHERE FirstName='Peter'");

while($row = mysqli_fetch_array($result))
{
echo $row['FirstName'] . " " . $row['LastName'];
echo "<br>";
}
?>
```

以上代码将输出：

```
Peter Griffin
```

PHP MySQL Order By 关键词

ORDER BY 关键词用于对记录集中的数据进行排序。

ORDER BY 关键词

ORDER BY 关键词用于对记录集中的数据进行排序。

ORDER BY 关键词默认对记录进行升序排序。

如果你想降序排序，请使用 DESC 关键字。

语法

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s) ASC|DESC
```

如需学习更多关于 SQL 的知识，请访问我们的 SQL 教程。

实例

下面的实例选取 "Persons" 表中存储的所有数据，并根据 "Age" 列对结果进行排序：

```
<?php
$con=mysqli_connect("example.com","peter","abc123","my_db");
// Check connection
if (mysqli_connect_errno())
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$result = mysqli_query($con,"SELECT * FROM Persons ORDER BY age");

while($row = mysqli_fetch_array($result))
{
echo $row['FirstName'];
echo " " . $row['LastName'];
echo " " . $row['Age'];
echo "<br>";
}

mysqli_close($con);
?>
```

以上结果将输出：

```
Glenn Quagmire 33
Peter Griffin 35
```

根据两列进行排序

可以根据多个列进行排序。当按照多个列进行排序时，只有第一列的值相同时才使用第二列：

```
SELECT column_name(s)
FROM table_name
ORDER BY column1, column2
```

PHP MySQL Update

UPDATE 语句用于中修改数据库表中的数据。

更新数据库中的数据

UPDATE 语句用于更新数据库表中已存在的记录。

语法

```
UPDATE table_name
SET column1=value, column2=value2,...
WHERE some_column=some_value
```

注释：请注意 UPDATE 语法中的 WHERE 子句。WHERE 子句规定了哪些记录需要更新。如果您想省去 WHERE 子句，所有的记录都会被更新！

如需学习更多关于 SQL 的知识，请访问我们的 SQL 教程。

为了让 PHP 执行上面的语句，我们必须使用 mysqli_query() 函数。该函数用于向 MySQL 连接发送查询或命令。

实例

在本教程的前面章节中，我们创建了一个名为 "Persons" 的表，如下所示：

FirstName	LastName	Age
Peter	Griffin	35
Glenn	Quagmire	33

下面的例子更新 "Persons" 表的一些数据：

```
<?php
$con=mysqli_connect("example.com","peter","abc123","my_db");
// Check connection
if (mysqli_connect_errno())
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

mysqli_query($con,"UPDATE Persons SET Age=36
WHERE FirstName='Peter' AND LastName='Griffin'");

mysqli_close($con);
?>
```

在这次更新后，"Persons" 表如下所示：

FirstName	LastName	Age
Peter	Griffin	36
Glenn	Quagmire	33

PHP MySQL Delete

DELETE 语句用于从数据库表中删除行。

删除数据库中的数据

DELETE FROM 语句用于从数据库表中删除记录。

语法

```
DELETE FROM table_name
WHERE some_column = some_value
```

注释：请注意 DELETE 语法中的 WHERE 子句。WHERE 子句规定了哪些记录需要删除。如果您想省去 WHERE 子句，所有的记录都会被删除！

如需学习更多关于 SQL 的知识，请访问我们的 SQL 教程。

为了让 PHP 执行上面的语句，我们必须使用 mysqli_query() 函数。该函数用于向 MySQL 连接发送查询或命令。

实例

请看下面的 "Persons" 表：

FirstName	LastName	Age
Peter	Griffin	35
Glenn	Quagmire	33

下面的实例删除 "Persons" 表中所有 LastName='Griffin' 的记录：

```
<?php
$con=mysqli_connect("example.com","peter","abc123","my_db");
// Check connection
if (mysqli_connect_errno())
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

mysqli_query($con,"DELETE FROM Persons WHERE LastName='Griffin'");

mysqli_close($con);
?>
```

在这次删除后，"Persons" 表如下所示：

FirstName	LastName	Age
Glenn	Quagmire	33

PHP 数据库 ODBC

ODBC 是一种应用程序编程接口（Application Programming Interface，API），使我们有能力连接到某个数据源（比如一个 MS Access 数据库）。

创建 ODBC 连接

通过一个 ODBC 连接，您可以连接到您的网络中的任何计算机上的任何数据库，只要 ODBC 连接是可用的。

这是创建到达 MS Access 数据库的 ODBC 连接的方法：

1. 在控制面板中打开管理工具图标。
2. 双击其中的数据源**(ODBC)**图标。
3. 选择系统 **DSN** 选项卡。
4. 点击系统 DSN 选项卡中的添加。
5. 选择**Microsoft Access Driver**。点击完成。
6. 在下一个界面，点击选择来定位数据库。
7. 为数据库起一个数据源名**(DSN)**。
8. 点击确定。

请注意，必须在您的网站所在的计算机上完成这个配置。如果您的计算机上正在运行 Internet 信息服务(IIS)，上面的指令将会生效，但是如果您的网站位于远程服务器，您必须拥有对该服务器的物理访问权限，或者请您的主机提供商为您建立 DSN。

连接到 ODBC

odbc_connect() 函数用于连接到 ODBC 数据源。该函数有四个参数：数据源名、用户名、密码以及可选的指针类型。

odbc_exec() 函数用于执行 SQL 语句。

实例

下面的实例创建了到达名为 northwind 的 DSN 的连接，没有用户名和密码。然后创建并执行一条 SQL 语句：

```
$conn=odbc_connect('northwind','','');
$sql="SELECT * FROM customers";
$rs=odbc_exec($conn,$sql);
```

取回记录

odbc_fetch_row() 函数用于从结果集中返回记录。如果能够返回行，则函数返回 true，否则返回 false。

该函数有两个参数：ODBC 结果标识符和可选的行号：

```
odbc_fetch_row($rs)
```

从记录中取回字段

odbc_result() 函数用于从记录中读取字段。该函数有两个参数：ODBC 结果标识符和字段编号或名称。

下面的代码行从记录中返回第一个字段的值：

```
$compname=odbc_result($rs,1);
```

下面的代码行返回名为 "CompanyName" 的字段的值：

```
$compname=odbc_result($rs,"CompanyName");
```

关闭 ODBC 连接

odbc_close() 函数用于关闭 ODBC 连接。

```
odbc_close($conn);
```

ODBC 实例

下面的实例展示了如何首先创建一个数据库连接，接着创建一个结果集，然后在 HTML 表格中显示数据。

```
<html>
<body>

<?php
$conn=odbc_connect('northwind','','');
if (!$conn)
{exit("Connection Failed: " . $conn);}
$sql="SELECT * FROM customers";
$rs=odbc_exec($conn,$sql);
if (!$rs)
{exit("Error in SQL");}
echo "<table><tr>";
echo "<th>Companyname</th>";
echo "<th>Contactname</th></tr>";
while (odbc_fetch_row($rs))
{
$compname=odbc_result($rs,"CompanyName");
$conname=odbc_result($rs,"ContactName");
echo "<tr><td>$compname</td>";
echo "<td>$conname</td></tr>";
}
odbc_close($conn);
echo "</table>";
?>

</body>
</html>
```

PHP XML

PHP XML Expat 解析器

内建的 **Expat** 解析器使在 **PHP** 中处理 **XML** 文档成为可能。

什么是 XML？

XML 用于描述数据，其焦点是数据是什么。XML 文件描述了数据的结构。

在 XML 中，没有预定义的标签。您必须定义自己的标签。

如果希望学习更多有关 XML 的内容，请访问我们的 XML 教程。

什么是 Expat？

如需读取和更新 - 创建创建并处理 - 一个 XML 文档，您需要 XML 解析器。

有两种基本的 XML 解析器类型：

- 基于树的解析器：这种解析器把 XML 文档转换为树型结构。它分析整篇文档，并提供了 API 来访问树种的元素，例如文档对象模型 (DOM)。
- 基于事件的解析器：将 XML 文档视为一系列的事件。当某个具体的事件发生时，解析器会调用函数来处理。

Expat 解析器是基于事件的解析器。

基于事件的解析器集中在 XML 文档的内容，而不是它们的结果。正因如此，基于事件的解析器能够比基于树的解析器更快地访问数据。

请看下面的 XML 片段：

```
<from>John</from>
```

基于事件的解析器把上面的 XML 报告为一连串的三个事件：

- 开始元素：from
- 开始 CDATA 部分, 值：John
- 关闭元素： from

上面的 XML 范例包含了形式良好的 XML。不过这个例子是无效的 XML，因为没有与它关联的文档类型声明 (DTD)，也没有内嵌的 DTD。

不过，在使用 Expat 解析器时，这没有区别。Expat 是不检查有效性的解析器，忽略任何 DTD。

作为一款基于事件、非验证的 XML 解析器，Expat 快速且轻巧，十分适合 PHP 的 web 应用程序。

注释：XML 文档必须形式良好，否则 Expat 会生成错误。

安装

XML Expat 解析器是 PHP 核心的组成部分。无需安装就可以使用这些函数。

XML 文件

将在我们的例子中使用下面的 XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

初始化 XML 解析器

我们要在 PHP 中初始化 XML 解析器，为不同的 XML 事件定义处理器，然后解析这个 XML 文件。

例子

```
<?php

//Initialize the XML parser
$parser=xml_parser_create();

//Function to use at the start of an element
function start($parser,$element_name,$element_attrs)
  {
  switch($element_name)
    {
    case "NOTE":
    echo "-- Note --<br />";
    break;
    case "TO":
    echo "To: ";
    break;
    case "FROM":
    echo "From: ";
    break;
    case "HEADING":
    echo "Heading: ";
    break;
    case "BODY":
    echo "Message: ";
    }
  }

//Function to use at the end of an element
function stop($parser,$element_name)
  {
  echo "<br />";
  }

//Function to use when finding character data
function char($parser,$data)
  {
  echo $data;
  }

//Specify element handler
xml_set_element_handler($parser,"start","stop");

//Specify data handler
xml_set_character_data_handler($parser,"char");

//Open XML file
$fp=fopen("test.xml","r");

//Read data
while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

//Free the XML parser
xml_parser_free($parser);

?>
```

以上代码的输出：

```
-- Note --
To: George
From: John
Heading: Reminder
Message: Don't forget the meeting!
```

工作原理解释：

- 通过 xml_parser_create() 函数初始化 XML 解析器
- 创建配合不同事件处理程序的的函数
- 添加 xml_set_element_handler() 函数来定义，当解析器遇到开始和结束标签时执行哪个函数
- 添加 xml_set_character_data_handler() 函数来定义，当解析器遇到字符数据时执行哪个函数
- 通过 xml_parse() 函数来解析文件 "test.xml"
- 万一有错误的话，添加 xml_error_string() 函数把 XML 错误转换为文本说明
- 调用 xml_parser_free() 函数来释放分配给 xml_parser_create() 函数的内存

更多 PHP Expat 解析器的信息

如需更多有关 PHP Expat 函数的信息，请访问我们的 PHP XML Parser 参考手册。

PHP XML DOM

内建的 **DOM** 解析器使在 **PHP** 中处理 **XML** 文档成为可能。

什么是 DOM？

W3C DOM 提供了针对 HTML 和 XML 文档的标准对象集，以及用于访问和操作这些文档的标准接口。

W3C DOM 被分为不同的部分 (Core, XML 和 HTML) 和不同的级别 (DOM Level 1/2/3)：

- Core DOM - 为任何结构化文档定义标准的对象集
- XML DOM - 为 XML 文档定义标准的对象集
- HTML DOM - 为 HTML 文档定义标准的对象集

如果您希望学习更多有关 XML DOM 的知识，请访问我们的 XML DOM 教程。

XML 解析

如需读取和更新 - 创建创建并处理 - 一个 XML 文档，您需要 XML 解析器。

有两种基本的 XML 解析器类型：

- 基于树的解析器：这种解析器把 XML 文档转换为树型结构。它分析整篇文档，并提供了 API 来访问树种的元素，例如文档对象模型 (DOM)。
- 基于事件的解析器：将 XML 文档视为一系列的事件。当某个具体的事件发生时，解析器会调用函数来处理。

DOM 解析器是基于树的解析器。

请看下面的 XML 文档片段：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<from>John</from>
```

XML DOM 把 XML 视为一个树形结构：

- Level 1: XML 文档
- Level 2: 根元素: <from>
- Level 3: 文本元素: "John"

安装

DOM XML 解析器函数是 PHP 核心的组成部分。无需安装就可以使用这些函数。

XML 文件

将在我们的例子中使用下面的 XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

加载和输出 XML

我们需要初始化 XML 解析器，加载 XML，并把它输出：

例子

```
<?php
$xmlDoc = new DOMDocument();
$xmlDoc->load("note.xml");

print $xmlDoc->saveXML();
?>
```

以上代码的输出：

```
George John Reminder Don't forget the meeting!
```

假如您在浏览器窗口中查看源代码，会看到下面这些 HTML：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

上面的例子创建了一个 DOMDocument-Object，并把 "note.xml" 中的 XML 载入这个文档对象中。

saveXML() 函数把内部 XML 文档放入一个字符串，这样我们就可以输出它。

循环 XML

我们要初始化 XML 解析器，加载 XML，并循环 <note> 元素的所有元素：

例子

```
<?php
$xmlDoc = new DOMDocument();
$xmlDoc->load("note.xml");

$x = $xmlDoc->documentElement;
foreach ($x->childNodes AS $item)
  {
  print $item->nodeName . " = " . $item->nodeValue . "<br />";
  }
?>
```

以上代码的输出：

```
#text =
to = George
#text =
from = John
#text =
heading = Reminder
#text =
body = Don't forget the meeting!
#text =
```

在上面的例子中，您看到了每个元素之间存在空的文本节点。

当 XML 生成时，它通常会在节点之间包含空白。XML DOM 解析器把它们当作普通的元素，如果您不注意它们，有时会产生问题。

如果您希望学习更多有关 XML DOM 的知识，请访问我们的 XML DOM 教程。

PHP SimpleXML

SimpleXML 处理最普通的 **XML** 任务，其余的任务则交由其它扩展。

什么是 SimpleXML？

SimpleXML 是 PHP 5 中的新特性。在了解 XML 文档 layout 的情况下，它是一种取得元素属性和文本的便利途径。

与 DOM 或 Expat 解析器相比，SimpleXML 仅仅用几行代码就可以从元素中读取文本数据。

SimpleXML 可把 XML 文档转换为对象，比如：

- 元素 - 被转换为 SimpleXMLElement 对象的单一属性。当同一级别上存在多个元素时，它们会被置于数组中。
- 属性 - 通过使用关联数组进行访问，其中的下标对应属性名称。
- 元素数据 - 来自元素的文本数据被转换为字符串。如果一个元素拥有多个文本节点，则按照它们被找到的顺序进行排列。

当执行类似下列的基础任务时，SimpleXML 使用起来非常快捷：

- 读取 XML 文件
- 从 XML 字符串中提取数据
- 编辑文本节点或属性

不过，在处理高级 XML 时，比如命名空间，最好使用 Expat 解析器或 XML DOM。

安装

从 PHP 5.0 开始，SimpleXML 函数是 PHP 核心的组成部分。无需安装就可以使用这些函数。

使用 SimpleXML

下面是 XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

我们打算从上面的 XML 文件输出元素的名称和数据。

这是需要做的事情：

1. 加载 XML 文件
2. 取得第一个元素的名称
3. 使用 children() 函数创建在每个子节点上触发的循环
4. 输出每个子节点的元素名称和数据

例子

```
<?php
$xml = simplexml_load_file("test.xml");

echo $xml->getName() . "<br />";

foreach($xml->children() as $child)
  {
  echo $child->getName() . ": " . $child . "<br />";
  }
?>
```

以上代码的输出：

```
note
to: George
from: John
heading: Reminder
body: Don't forget the meeting!
```

更多有关 PHP SimpleXML 的信息

如需更多有关 PHP SimpleXML 的信息，请访问我们的 PHP SimpleXML 参考手册。

PHP AJAX

AJAX 简介

AJAX = Asynchronous JavaScript And XML（异步 JavaScript 及 XML）

AJAX 是 _A_synchronous _J_avaScript _A_nd _X_ML 的首字母缩写。

AJAX 并不是一种新的编程语言，而仅仅是一种新的技术，它可以创建更好、更快且交互性更强的 web 应用程序。

AJAX 使用 JavaScript 在 web 浏览器与 web 服务器之间来发送和接收数据。

通过在幕后与 web 服务器交换数据，而不是每当用户作出改变时重载整个 web 页面，AJAX 技术可以使网页更迅速地响应。

AJAX 基于开放的标准

AJAX 基于以下开放的标准：

- *JavaScript*
- *XML*
- *HTML*
- *CSS*

在 AJAX 中使用的开放标准被良好地定义，并得到所有主要浏览器的支持。AJAX 应用程序独立于浏览器和平台。（可以说，它是一种跨平台跨浏览器的技术）。

AJAX 事关更好的 Internet 应用程序

与桌面应用程序相比，Web 应用程序有很多优势：

- 可拥有更多用户
- 更容易安装和维护
- 更容易开发

但是， 应用程序不总是象传统应用程序那样强大和友好。

通过 AJAX，可以使 Internet 应用程序更加强大（更轻巧、更快速，且更易使用）。

今天您就可以开始使用 AJAX

没有什么新知识需要学习。

AJAX 基于开放的标准。而这些标准已被大多数开发者使用多年。

大多数 web 应用程序可通过使用 AJAX 技术进行重写，来替代传统的 HTML 表单。

AJAX 使用 XML 和 HTTP 请求

传统的 web 应用程序会把数据提交到 web 服务器（使用 HTML 表单）。在 web 服务器把数据处理完毕之后，会向用户返回一张完整的新网页。

由于每当用户提交输入，服务器就会返回新网页，传统的 web 应用程序往往运行缓慢，且越来越不友好。

通过 AJAX，web 应用程序无需重载网页，就可以发送并取回数据。完成这项工作，需要通过向服务器发送 HTTP 请求（在幕后），并通过当服务器返回数据时使用 JavaScript 仅仅修改网页的某部分。

一般使用 XML 作为接收服务器数据的格式，尽管可以使用任何格式，包括纯文本。

您将在本教程接下来的章节学习到如何完成这些工作。

PHP 和 AJAX

不存在什么 AJAX 服务器。

AJAX 是一种在浏览器运行的技术。它使用浏览器与 web 服务器之间的异步数据传输，使网页从服务器请求少量的信息，而不是整张页面。

AJAX 是一种独立于 web 服务器软件的 web 浏览器技术。

但是，在本教程中，我们将集中在运行在 PHP 服务器上的实际案例，而不是 AJAX 的工作原理。

如需阅读更多有关 AJAX 如何工作的知识，请访问我们的 AJAX 教程。

AJAX XMLHttpRequest

XMLHttpRequest 对象使 **AJAX** 成为可能。

XMLHttpRequest

XMLHttpRequest 对象是 AJAX 的关键。

该对象在 Internet Explorer 5.5 与 2000 年 7 月发布之后就已经可用了，但是在 2005 人们开始讨论 AJAX 和 Web 2.0 之前，这个对象并没有得到充分的认识。

创建 XMLHttpRequest 对象

不同的浏览器使用不同的方法来创建 *XMLHttpRequest* 对象。

Internet Explorer 使用 *ActiveXObject*。

其他浏览器使用名为 *XMLHttpRequest* 的 JavaScript 内建对象。

要克服这个问题，可以使用这段简单的代码：

```
var XMLHttp=null
if (window.XMLHttpRequest)
  {
  XMLHttp=new XMLHttpRequest()
  }
else if (window.ActiveXObject)
  {
  XMLHttp=new ActiveXObject("Microsoft.XMLHTTP")
  }
```

代码解释：

1. 首先创建一个作为 XMLHttpRequest 对象使用的 *XMLHttp* 变量。把它的值设置为 null。
2. 然后测试 *window.XMLHttpRequest* 对象是否可用。在新版本的 Firefox, Mozilla, Opera 以及 Safari 浏览器中，该对象是可用的。
3. 如果可用，则用它创建一个新对象：*XMLHttp=new XMLHttpRequest()*
4. 如果不可用，则检测 *window.ActiveXObject* 是否可用。在 Internet Explorer version 5.5 及更高的版本中，该对象是可用的。
5. 如果可用，使用它来创建一个新对象：*XMLHttp=new ActiveXObject()*

改进的例子

一些程序员喜欢使用最新最快的版本的 XMLHttpRequest 对象。

下面的例子试图加载微软最新版本的 "Msxml2.XMLHTTP"，在 Internet Explorer 6 中可用，如果无法加载，则后退到 "Microsoft.XMLHTTP"，在 Internet Explorer 5.5 及其后版本中可用。

```
function GetXmlHttpObject()
{
var xmlHttp=null;

try
 {
 // Firefox, Opera 8.0+, Safari
 xmlHttp=new XMLHttpRequest();
 }
catch (e)
 {
 // Internet Explorer
 try
  {
  xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
  }
 catch (e)
  {
  xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
 }
return xmlHttp;
}
```

代码解释：

1. 首先创建用作 XMLHttpRequest 对象的 *XMLHttp* 变量。把它的值设置为 null。
2. 按照 web 标准创建对象 (Mozilla, Opera 以及 Safari)：*XMLHttp=new XMLHttpRequest()*
3. 按照微软的方式创建对象，在 Internet Explorer 6 及更高的版本可用：*XMLHttp=new ActiveXObject("Msxml2.XMLHTTP")*
4. 如果捕获错误，则尝试更老的方法 (Internet Explorer 5.5) ：*XMLHttp=new ActiveXObject("Microsoft.XMLHTTP")*

更多有关 **XMLHttpRequest** 对象的信息

如果您希望阅读更多有关 XMLHttpRequest 的内容，请访问我们的 AJAX 教程。

PHP 和 AJAX 请求

- XMLHttpRequest

AJAX 请求

在下面的 AJAX 例子中，我们将演示当用户向 web 表单中输入数据时，网页如何与在线的 web 服务器进行通信。

这个例子包括三张页面：

- 一个简单的 HTML 表单
- 一段 JavaScript
- 一张 PHP 页面

HTML 表单

这是 HTML 表单。它包含一个简单的 HTML 表单和指向 JavaScript 的链接：

```
<html>
<head>
<script src="clienthint.js"></script>
</head>

<body>

<form>
First Name:
<input type="text" id="txt1"
onkeyup="showHint(this.value)">
</form>

<p>Suggestions: <span id="txtHint"></span></p>

</body>
</html>
```

例子解释 - HTML 表单

正如您看到的，上面的 HTML 页面含有一个简单的 HTML 表单，其中带有一个名为 "txt1" 的输入字段。

该表单是这样工作的：

1. 当用户在输入域中按下并松开按键时，会触发一个事件
2. 当该事件被触发时，执行名为 showHint() 的函数

3. 表单的下面是一个名为 "txtHint" 的 <span>。它用作 showHint() 函数所返回数据的占位符。

JavaScript

JavaScript 代码存储在 "clienthint.js" 文件中，它被链接到 HTML 文档：

```
var xmlHttp

function showHint(str)
{
if (str.length==0)
  {
  document.getElementById("txtHint").innerHTML=""
  return
  }
xmlHttp=GetXmlHttpObject()
if (xmlHttp==null)
  {
  alert ("Browser does not support HTTP Request")
  return
  }
var url="gethint.php"
url=url+"?q="+str
url=url+"&sid="+Math.random()
xmlHttp.onreadystatechange=stateChanged
xmlHttp.open("GET",url,true)
xmlHttp.send(null)
}

function stateChanged()
{
if (xmlHttp.readyState==4 || xmlHttp.readyState=="complete")
 {
 document.getElementById("txtHint").innerHTML=xmlHttp.responseText
 }
}

function GetXmlHttpObject()
{
var xmlHttp=null;
try
 {
 // Firefox, Opera 8.0+, Safari
 xmlHttp=new XMLHttpRequest();
 }
catch (e)
 {
 // Internet Explorer
 try
  {
  xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
  }
 catch (e)
  {
  xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
 }
return xmlHttp;
}
```

例子解释：

showHint() 函数

每当在输入域中输入一个字符，该函数就会被执行一次。

如果文本框中有内容 (str.length > 0)，该函数这样执行：

1. 定义要发送到服务器的 URL（文件名）
2. 把带有输入域内容的参数 (q) 添加到这个 URL
3. 添加一个随机数，以防服务器使用缓存文件
4. 调用 GetXmlHttpObject 函数来创建 XMLHTTP 对象，并在事件被触发时告知该对象执行名为 stateChanged 的函数
5. 用给定的 URL 来打开打开这个 XMLHTTP 对象
6. 向服务器发送 HTTP 请求

如果输入域为空，则函数简单地清空 txtHint 占位符的内容。

stateChanged() 函数

每当 XMLHTTP 对象的状态发生改变，则执行该函数。

在状态变成 4 （或 "complete"）时，用响应文本填充 txtHint 占位符 txtHint 的内容。

GetXmlHttpObject() 函数

AJAX 应用程序只能运行在完整支持 XML 的 web 浏览器中。

上面的代码调用了名为 GetXmlHttpObject() 的函数。

该函数的作用是解决为不同浏览器创建不同 XMLHTTP 对象的问题。

这一点在上一节中已经解释过了。

PHP 页面

被 JavaScript 代码调用的服务器页面是一个名为 "gethint.php" 的简单服务器页面。

"gethint.php" 中的代码会检查名字数组，然后向客户端返回对应的名字：

```
<?php
// Fill up array with names
$a[]="Anna";
$a[]="Brittany";
$a[]="Cinderella";
$a[]="Diana";
$a[]="Eva";
$a[]="Fiona";
$a[]="Gunda";
$a[]="Hege";
$a[]="Inga";
$a[]="Johanna";
$a[]="Kitty";
$a[]="Linda";
$a[]="Nina";
$a[]="Ophelia";
$a[]="Petunia";
$a[]="Amanda";
$a[]="Raquel";
$a[]="Cindy";
$a[]="Doris";
$a[]="Eve";
$a[]="Evita";
$a[]="Sunniva";
$a[]="Tove";
$a[]="Unni";
$a[]="Violet";
$a[]="Liza";
$a[]="Elizabeth";
$a[]="Ellen";
$a[]="Wenche";
$a[]="Vicky";

//get the q parameter from URL
$q=$_GET["q"];

//lookup all hints from array if length of q>0
if (strlen($q) > 0)
{
$hint="";
for($i=0; $i<count($a); $i++)
  {
  if (strtolower($q)==strtolower(substr($a[$i],0,strlen($q))))
    {
    if ($hint=="")
      {
      $hint=$a[$i];
      }
    else
      {
      $hint=$hint." , ".$a[$i];
      }
    }
  }
}

//Set output to "no suggestion" if no hint were found
//or to the correct values
if ($hint == "")
{
$response="no suggestion";
}
else
{
$response=$hint;
}

//output the response
echo $response;
?>
```

如果存在从 JavaScript 送来的文本 (strlen($q) > 0)，则：

1. 找到与 JavaScript 所传送的字符相匹配的名字
2. 如果找到多个名字，把所有名字包含在 response 字符串中
3. 如果没有找到匹配的名字，把 response 设置为 "no suggestion"
4. 如果找到一个或多个名字，把 response 设置为这些名字
5. 把 response 发送到 "txtHint" 占位符

6. XMLHttpRequest

PHP 和 AJAX XML 实例

AJAX 可与 **XML** 文件进行交互式通信。

AJAX XML 实例

在下面的 AJAX 实例中，我们将演示网页如何使用 AJAX 技术从 XML 文件中读取信息。

本例包括三张页面：

- 一个简单 HTML 表单
- 一个 XML 文件
- 一个 JavaScript 文件
- 一张 PHP 页面

HTML 表单

上面的例子包含了一张简单的 HTML 表单，以及指向 JavaScript 的链接：

```
<html>
<head>
<script src="selectcd.js"></script>
</head>

<body>

<form>
Select a CD:
<select name="cds" onchange="showCD(this.value)">
<option value="Bob Dylan">Bob Dylan</option>
<option value="Bee Gees">Bee Gees</option>
<option value="Cat Stevens">Cat Stevens</option>
</select>
</form>

<p>
<div id="txtHint"><b>CD info will be listed here.</b></div>
</p>

</body>
</html>
```

例子解释：

正如您看到的，它仅仅是一张简单的 HTML 表单，其中带有名为 "cds" 的下拉列表。

表单下面的段落包含了一个名为 "txtHint" 的 div。这个 div 用作从 web 服务器检索到的数据的占位符。

当用户选择数据时，会执行名为 "showCD" 的函数。这个函数的执行是由 "onchange" 事件触发的。

换句话说，每当用户改变了下拉列表中的值，就会调用 showCD 函数。

XML 文件

XML 文件是 "cd_catalog.xml"。该文件中包含了有关 CD 收藏的数据。

JavaScript

这是存储在 "selectcd.js" 文件中的 JavaScript 代码：

```
var xmlHttp

function showCD(str)
{
xmlHttp=GetXmlHttpObject()
if (xmlHttp==null)
 {
 alert ("Browser does not support HTTP Request")
 return
 }
var url="getcd.php"
url=url+"?q="+str
url=url+"&sid="+Math.random()
xmlHttp.onreadystatechange=stateChanged
xmlHttp.open("GET",url,true)
xmlHttp.send(null)
}

function stateChanged()
{
 if (xmlHttp.readyState==4 || xmlHttp.readyState=="complete")
 {
 document.getElementById("txtHint").innerHTML=xmlHttp.responseText
 }
}

function GetXmlHttpObject()
{
var xmlHttp=null;

try
 {
 // Firefox, Opera 8.0+, Safari
 xmlHttp=new XMLHttpRequest();
 }
catch (e)
 {
 // Internet Explorer
 try
  {
  xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
  }
 catch (e)
  {
  xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
 }
return xmlHttp;
}
```

例子解释：

stateChanged() 和 GetXmlHttpObject 函数与上一节中的相同，您可以参阅上一页中的相关解释。

showCD() 函数

假如选择了下拉列表中的某个项目，则函数执行：

1. 调用 GetXmlHttpObject 函数来创建 XMLHTTP 对象
2. 定义发送到服务器的 URL（文件名）
3. 向 URL 添加带有下拉列表内容的参数 (q)

4. 添加一个随机数，以防服务器使用缓存的文件
5. 当触发事件时调用 stateChanged
6. 通过给定的 URL 打开 XMLHTTP 对象
7. 向服务器发送 HTTP 请求

PHP 页面

这个被 JavaScript 调用的服务器页面，是一个名为 "getcd.php" 的简单 PHP 文件。

这张页面是用 PHP 编写的，使用 XML DOM 来加载 XML 文档 "cd_catalog.xml"。

代码运行针对 XML 文件的查询，并以 HTML 返回结果：

```
<?php
$q=$_GET["q"];

$xmlDoc = new DOMDocument();
$xmlDoc->load("cd_catalog.xml");

$x=$xmlDoc->getElementsByTagName('ARTIST');

for ($i=0; $i<=$x->length-1; $i++)
{
//Process only element nodes
if ($x->item($i)->nodeType==1)
  {
  if ($x->item($i)->childNodes->item(0)->nodeValue == $q)
    {
    $y=($x->item($i)->parentNode);
    }
  }
}

$cd=($y->childNodes);

for ($i=0;$i<$cd->length;$i++)
{
//Process only element nodes
if ($cd->item($i)->nodeType==1)
  {
  echo($cd->item($i)->nodeName);
  echo(": ");
  echo($cd->item($i)->childNodes->item(0)->nodeValue);
  echo("<br />");
  }
}
?>
```

例子解释

当请求从 JavaScript 发送到 PHP 页面时，发生：

1. PHP 创建 "cd_catalog.xml" 文件的 XML DOM 对象
2. 循环所有 "artist" 元素 (nodetypes = 1)，查找与 JavaScript 所传数据向匹配的名字
3. 找到 CD 包含的正确 artist
4. 输出 album 的信息，并发送到 "txtHint" 占位符

PHP 和 AJAX MySQL 数据库实例

AJAX 可用来与数据库进行交互式通信。

AJAX 数据库实例

在下面的 AJAX 实例中，我们将演示网页如何使用 AJAX 技术从 MySQL 数据库中读取信息。

此列由四个元素组成：

- MySQL 数据库
- 简单的 HTML 表单
- JavaScript
- PHP 页面

数据库

将在本例中使用的数据库看起来类似这样：

id	FirstName	LastName	Age	Hometown	Job
1	Peter	Griffin	41	Quahog	Brewery
2	Lois	Griffin	40	Newport	Piano Teacher
3	Joseph	Swanson	39	Quahog	Police Officer
4	Glenn	Quagmire	41	Quahog	Pilot

HTML 表单

上面的例子包含了一个简单的 HTML 表单，以及指向 JavaScript 的链接：

```
<html>
<head>
<script src="selectuser.js"></script>
</head>
<body>

<form>
Select a User:
<select name="users" onchange="showUser(this.value)">
<option value="1">Peter Griffin</option>
<option value="2">Lois Griffin</option>
<option value="3">Glenn Quagmire</option>
<option value="4">Joseph Swanson</option>
</select>
</form>

<p>
<div id="txtHint"><b>User info will be listed here.</b></div>
</p>

</body>
</html>
```

例子解释 - HTML 表单

正如您看到的，它仅仅是一个简单的 HTML 表单，其中带有名为 "users" 的下拉列表，这个列表包含了姓名，以及与数据库的 "id" 对应的选项值。

表单下面的段落包含了名为 "txtHint" 的 div。这个 div 用作从 web 服务器检索到的信息的占位符。

当用户选择数据时，执行名为 "showUser()" 的函数。该函数的执行由 "onchange" 事件触发。

换句话说：每当用户改变下拉列表中的值，就会调用 showUser() 函数。

JavaScript

这是存储在 "selectuser.js" 文件中的 JavaScript 代码：

```
var xmlHttp

function showUser(str)
{
xmlHttp=GetXmlHttpObject()
if (xmlHttp==null)
 {
 alert ("Browser does not support HTTP Request")
 return
 }
var url="getuser.php"
url=url+"?q="+str
url=url+"&sid="+Math.random()
xmlHttp.onreadystatechange=stateChanged
xmlHttp.open("GET",url,true)
xmlHttp.send(null)
}

function stateChanged()
{
if (xmlHttp.readyState==4 || xmlHttp.readyState=="complete")
 {
 document.getElementById("txtHint").innerHTML=xmlHttp.responseText
 }
}

function GetXmlHttpObject()
{
var xmlHttp=null;
try
 {
 // Firefox, Opera 8.0+, Safari
 xmlHttp=new XMLHttpRequest();
 }
catch (e)
 {
 //Internet Explorer
 try
  {
  xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
  }
 catch (e)
  {
  xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
 }
return xmlHttp;
}
```

例子解释：

stateChanged() 和 GetXmlHttpObject 函数与 PHP AJAX 请求 那一节中的相同，您可以参阅其中的相关解释。

showUser() 函数

假如下拉列表中的项目被选择，函数执行：

1. 调用 GetXmlHttpObject 函数来创建 XMLHTTP 对象
2. 定义发送到服务器的 URL（文件名）
3. 向 URL 添加带有下拉列表内容的参数 (q)

4. 添加一个随机数，以防服务器使用缓存的文件
5. 当触发事件时调用 stateChanged
6. 通过给定的 URL 打开 XMLHTTP 对象
7. 向服务器发送 HTTP 请求

PHP 页面

由 JavaScript 调用的服务器页面，是名为 "getuser.php" 的简单 PHP 文件。

该页面用 PHP 编写，并使用 MySQL 数据库。

其中的代码执行针对数据库的 SQL 查询，并以 HTML 表格返回结果：

```
<?php
$q=$_GET["q"];

$con = mysql_connect('localhost', 'peter', 'abc123');
if (!$con)
 {
 die('Could not connect: ' . mysql_error());
 }

mysql_select_db("ajax_demo", $con);

$sql="SELECT * FROM user WHERE id = '".$q."'";

$result = mysql_query($sql);

echo "<table border='1'>
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Age</th>
<th>Hometown</th>
<th>Job</th>
</tr>";

while($row = mysql_fetch_array($result))
 {
 echo "<tr>";
 echo "<td>" . $row['FirstName'] . "</td>";
 echo "<td>" . $row['LastName'] . "</td>";
 echo "<td>" . $row['Age'] . "</td>";
 echo "<td>" . $row['Hometown'] . "</td>";
 echo "<td>" . $row['Job'] . "</td>";
 echo "</tr>";
 }
echo "</table>";

mysql_close($con);
?>
```

例子解释：

当查询从 JavaScript 被发送到这个 PHP 页面，会发生：

1. PHP 打开到达 MySQL 服务器的连接

2. 找到拥有指定姓名的 "user"
3. 创建表格，插入数据，然后将其发送到 "txtHint" 占位符

PHP 和 AJAX responseXML 实例

AJAX 可用于以 **XML** 返回数据库信息。

AJAX Database 转 XML 实例 （测试说明：该实例功能未实现）

在下面的 AJAX 实例中，我们将演示网页如何从 MySQL 数据库中读取信息，把数据转换为 XML 文档，并在不同的地方使用这个文档来显示信息。

本例与上一节中的 "PHP AJAX Database" 这个例子很相似，不过有一个很大的不同：在本例中，我们通过使用 responseXML 函数从 PHP 页面得到的是 XML 形式的数据。

把 XML 文档作为响应来接收，使我们有能力更新页面的多个位置，而不仅仅是接收一个 PHP 输出并显示出来。

在本例中，我们将使用从数据库接收到的信息来更新多个 <span> 元素。

此列由四个元素组成：

- MySQL 数据库
- 简单的 HTML 表单
- JavaScript
- PHP 页面

数据库

将在本例中使用的数据库看起来类似这样：

id	FirstName	LastName	Age	Hometown	Job
1	Peter	Griffin	41	Quahog	Brewery
2	Lois	Griffin	40	Newport	Piano Teacher
3	Joseph	Swanson	39	Quahog	Police Officer
4	Glenn	Quagmire	41	Quahog	Pilot

HTML 表单

上面的例子包含了一个简单的 HTML 表单，以及指向 JavaScript 的链接：

```
<html>
<head>
<script src="responsexml.js"></script>
</head>
<body>

<form>
Select a User:
<select name="users" onchange="showUser(this.value)">
<option value="1">Peter Griffin</option>
<option value="2">Lois Griffin</option>
<option value="3">Glenn Quagmire</option>
<option value="4">Joseph Swanson</option>
</select>
</form>

<h2>
<span id="firstname"></span> <span id="lastname"></span>
</h2>

<span id="job"></span>

<div style="text-align: right">
<span id="age_text"></span>
<span id="age"></span>
<span id="hometown_text"></span>
<span id="hometown"></span>
</div>

</body>
</html>
```

例子解释 - HTML 表单

- HTML 表单是一个下拉列表，其 name 属性的值是 "users"，可选项的值与数据库的 id 字段相对应
- 表单下面有几个 <span> 元素，它们用作我们所接收到的不同的值的占位符
- 当用户选择了具体的选项，函数 "showUser()" 就会执行。该函数的执行由 "onchange" 事件触发

换句话说，每当用户在下拉列表中改变了值，函数 showUser() 就会执行，并在指定的 <span> 元素中输出结果。

JavaScript

这是存储在文件 "responsexml.js" 中的 JavaScript 代码：

```
var xmlHttp

function showUser(str)
 {
 xmlHttp=GetXmlHttpObject()
 if (xmlHttp==null)
  {
  alert ("Browser does not support HTTP Request")
  return
  }
 var url="responsexml.php"
 url=url+"?q="+str
 url=url+"&sid="+Math.random()
 xmlHttp.onreadystatechange=stateChanged
 xmlHttp.open("GET",url,true)
 xmlHttp.send(null)
 }

function stateChanged()
{
if (xmlHttp.readyState==4 || xmlHttp.readyState=="complete")
{
 xmlDoc=xmlHttp.responseXML;
 document.getElementById("firstname").innerHTML=
 xmlDoc.getElementsByTagName("firstname")[0].childNodes[0].nodeValue;
 document.getElementById("lastname").innerHTML=
 xmlDoc.getElementsByTagName("lastname")[0].childNodes[0].nodeValue;
 document.getElementById("job").innerHTML=
 xmlDoc.getElementsByTagName("job")[0].childNodes[0].nodeValue;
 document.getElementById("age_text").innerHTML="Age: ";
 document.getElementById("age").innerHTML=
 xmlDoc.getElementsByTagName("age")[0].childNodes[0].nodeValue;
 document.getElementById("hometown_text").innerHTML="<br/>From: ";
 document.getElementById("hometown").innerHTML=
 xmlDoc.getElementsByTagName("hometown")[0].childNodes[0].nodeValue;
 }
}

function GetXmlHttpObject()
 {
 var objXMLHttp=null
 if (window.XMLHttpRequest)
  {
  objXMLHttp=new XMLHttpRequest()
  }
 else if (window.ActiveXObject)
  {
  objXMLHttp=new ActiveXObject("Microsoft.XMLHTTP")
  }
 return objXMLHttp
 }
```

例子解释：

showUser() 与 GetXmlHttpObject 函数与 PHP 和 AJAX MySQL 数据库实例 这一节中的例子是相同的。您可以参阅其中的相关解释。

stateChanged() 函数

如果选择了下拉列表中的项目，该函数执行：

1. 通过使用 responseXML 函数，把 "xmlDoc" 变量定义为一个 XML 文档
2. 从这个 XML 文档中取回数据，把它们放在正确的 "span" 元素中

PHP 页面

这个由 JavaScript 调用的服务器页面，是一个名为 "responsexml.php" 的简单的 PHP 文件。

该页面由 PHP 编写，并使用 MySQL 数据库。

代码会运行一段针对数据库的 SQL 查询，并以 XML 文档返回结果：

```
<?php
header('Content-Type: text/xml');
header("Cache-Control: no-cache, must-revalidate");
//A date in the past
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");

$q=$_GET["q"];

$con = mysql_connect('localhost', 'peter', 'abc123');
if (!$con)
 {
 die('Could not connect: ' . mysql_error());
 }

mysql_select_db("ajax_demo", $con);

$sql="SELECT * FROM user WHERE id = ".$q."";

$result = mysql_query($sql);

echo '<?xml version="1.0" encoding="ISO-8859-1"?>
<person>';
while($row = mysql_fetch_array($result))
 {
 echo "<firstname>" . $row['FirstName'] . "</firstname>";
 echo "<lastname>" . $row['LastName'] . "</lastname>";
 echo "<age>" . $row['Age'] . "</age>";
 echo "<hometown>" . $row['Hometown'] . "</hometown>";
 echo "<job>" . $row['Job'] . "</job>";
 }
echo "</person>";

mysql_close($con);
?>
```

例子解释：

当查询从 JavaScript 送达 PHP 页面时，会发生：

- PHP 文档的 content-type 被设置为 "text/xml"
- PHP 文档被设置为 "no-cache"，以防止缓存
- 用 HTML 页面送来的数据设置 $q 变量
- PHP 打开与 MySQL 服务器的连接
- 找到带有指定 id 的 "user"
- 以 XML 文档输出数据

PHP 和 AJAX Live Search

AJAX 可为用户提供更友好、交互性更强的搜索体验。

AJAX Live Search

在下面的 AJAX 例子中，我们将演示一个实时的搜索。

实时的搜索与传统搜索相比，具有很多优势：

- 当键入数据时，就会显示出匹配的结果
- 当继续键入数据时，对结果进行过滤
- 如果结果太少，删除字符就可以获得更宽的范围

本例包括四个元素：

- 简单的 HTML 表单
- JavaScript
- PHP 页面
- XML 文档

在本例中，结果在一个 XML 文档 (links.xml) 中进行查找。为了让这个例子小而简单，我们只提供 8 个结果。

HTML 表单

这是 HTML 页面。它包含一个简单的 HTML 表单，针对此表单的 CSS 样式，以及指向 JavaScript 的链接：

```
<html>
<head>
<script src="livesearch.js"></script>
<style type="text/css">
#livesearch
  {
  margin:0px;
  width:194px;
  }
#txt1
  {
  margin:0px;
  }
</style>
</head>
<body>

<form>
<input type="text" id="txt1" size="30"
onkeyup="showResult(this.value)">

<div id="livesearch"></div>
</form>

</body>
</html>
```

例子解释 - HTML 表单

正如你看到的，HTML 页面包含一个简单的 HTML 表单，其中的文本框名为 "txt1"。

表单是这样工作的：

1. 当用户在文本框中按键并松开按键时，会触发一个事件
2. 当事件触发时，会执行名为 showResult() 的函数
3. 表单下面是名为 "livesearch" 的 <div> 元素。它用作 showResult() 所返回数据的占位符

JavaScript

JavaScript 代码存储在与 HTML 文档连接的 "livesearch.js" 中：

```
var xmlHttp

function showResult(str)
{
if (str.length==0)
 {
 document.getElementById("livesearch").
 innerHTML="";
 document.getElementById("livesearch").
 style.border="0px";
 return
 }

xmlHttp=GetXmlHttpObject()

if (xmlHttp==null)
 {
 alert ("Browser does not support HTTP Request")
 return
 }

var url="livesearch.php"
url=url+"?q="+str
url=url+"&sid="+Math.random()
xmlHttp.onreadystatechange=stateChanged
xmlHttp.open("GET",url,true)
xmlHttp.send(null)
}

function stateChanged()
{
if (xmlHttp.readyState==4 || xmlHttp.readyState=="complete")
 {
 document.getElementById("livesearch").
 innerHTML=xmlHttp.responseText;
 document.getElementById("livesearch").
 style.border="1px solid #A5ACB2";
 }
}

function GetXmlHttpObject()
{
var xmlHttp=null;
try
 {
 // Firefox, Opera 8.0+, Safari
 xmlHttp=new XMLHttpRequest();
 }
catch (e)
 {
 // Internet Explorer
 try
  {
  xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
  }
 catch (e)
  {
  xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
 }
return xmlHttp;
}
```

例子解释：

GetXmlHttpObject 与 PHP 和 AJAX 请求 中的例子相同。

showResult() 函数

该函数每当一个字符输入文本框就会执行一次。

如果文本域中没有输入 (str.length == 0)，该函数把返回字段设置为空，并删除周围的任何边框。

不过，如果文本域中存在输入，则函数执行：

1. 定义发送到服务器的 url （文件名）
2. 把带有输入框内容的参数 (q) 添加到 url
3. 添加一个随机数，以防止服务器使用缓存文件
4. 调用 GetXmlHttpObject 函数来创建 XMLHTTP 对象，并在触发一个变化时告知此函数执行名为 stateChanged 的一个函数
5. 使用给定的 url 来打开 XMLHTTP 对象
6. 向服务器发送 HTTP 请求

stateChanged() 函数

每当 XMLHTTP 对象的状态发生变化时，该函数就会执行。

当状态变为 4 (或 "complete") 时，就会使用响应文本来填充 txtHint 占位符的内容，并在返回字段周围设置一个边框。

PHP 页面

由 JavaScript 代码调用的服务器页面是名为 "livesearch.php" 的 PHP 文件。

"livesearch.php" 中的代码检查那个 XML 文档 "links.xml"。该文档 w3school.com.cn 上的一些页面的标题和 URL。

这些代码会搜索 XML 文件中匹配搜索字符串的标题，并以 HTML 返回结果：

```
<?php
$xmlDoc = new DOMDocument();
$xmlDoc->load("links.xml");

$x=$xmlDoc->getElementsByTagName('link');

//get the q parameter from URL
$q=$_GET["q"];

//lookup all links from the xml file if length of q>0
if (strlen($q) > 0)
{
$hint="";
for($i=0; $i<($x->length); $i++)
 {
 $y=$x->item($i)->getElementsByTagName('title');
 $z=$x->item($i)->getElementsByTagName('url');
 if ($y->item(0)->nodeType==1)
  {
  //find a link matching the search text
  if (stristr($y->item(0)->childNodes->item(0)->nodeValue,$q))
   {
   if ($hint=="")
    {
    $hint="<a href='" .
    $z->item(0)->childNodes->item(0)->nodeValue .
    "' target='_blank'>" .
    $y->item(0)->childNodes->item(0)->nodeValue . "</a>";
    }
   else
    {
    $hint=$hint . "<br /><a href='" .
    $z->item(0)->childNodes->item(0)->nodeValue .
    "' target='_blank'>" .
    $y->item(0)->childNodes->item(0)->nodeValue . "</a>";
    }
   }
  }
 }
}

// Set output to "no suggestion" if no hint were found
// or to the correct values
if ($hint == "")
 {
 $response="no suggestion";
 }
else
 {
 $response=$hint;
 }

//output the response
echo $response;
?>
```

例子解释：

如果从 JavaScript 送来了任何文本 (strlen($q) > 0)，会发生：

1. PHP 创建 "links.xml" 文件的一个 XML DOM 对象
2. 遍历所有 "title" 元素 (nodetypes = 1)，以便找到匹配 JavaScript 所传数据的 name
3. 找到包含正确 title 的 link，并设置为 "$response" 变量。如果找到多于一个匹配，所有的匹配都会添加到变量

4. 如果没有找到匹配，则把 $response 变量设置为 "no suggestion"
5. $result 是送往 "livesearch" 占位符的输出

PHP 和 AJAX RSS 阅读器

RSS 阅读器用于阅读 **RSS Feed**。

RSS 允许对新闻和更新进行快速浏览。

AJAX RSS 阅读器

在下面的 AJAX 实例中，我们将演示一个 RSS 阅读器，通过它，来自 RSS 的内容在不进行刷新的情况下载入网页。

本例包括三个元素：

- 简单的 HTML 表单
- JavaScript
- PHP 页面

HTML 表单

这是 HTML 页面。它包含一个简单的 HTML 表单和执行一个 JavaScript 文件的链接：

```
<html>
<head>
<script type="text/javascript" src="getrss.js"></script>
</head>
<body>

<form>
Select an RSS-Feed:
<select onchange="showRSS(this.value)">
<option value="Google">Google News</option>
<option value="MSNBC">MSNBC News</option>
</select>
</form>

<p><div id="rssOutput">
<b>RSS Feed will be listed here.</b></div></p>
</body>
</html>
```

例子解释 - HTML 表单

正如您看到的，上面的 HTML 页面包含一个简单的 HTML 表单，其中带有一个下拉列表框。

表单是这样工作的：

1. 当用户选择下拉框中的选项时，会触发一个事件

2. 当事件触发时，执行 showRSS() 函数

表单下面是名为 "rssOutput" 的一个 <div>。它用作 showRSS() 函数所返回的数据的占位符。

JavaScript

JavaScript 代码存储在 "getrss.js" 中，它与 HTML 文档相连接：

```
var xmlHttp

function showRSS(str)
 {
 xmlHttp=GetXmlHttpObject()
 if (xmlHttp==null)
  {
  alert ("Browser does not support HTTP Request")
  return
  }
 var url="getrss.php"
 url=url+"?q="+str
 url=url+"&sid="+Math.random()
 xmlHttp.onreadystatechange=stateChanged
 xmlHttp.open("GET",url,true)
 xmlHttp.send(null)
 }

function stateChanged()
 {
 if (xmlHttp.readyState==4 || xmlHttp.readyState=="complete")
  {
  document.getElementById("rssOutput")
  .innerHTML=xmlHttp.responseText
  }
 }

function GetXmlHttpObject()
{
var xmlHttp=null;
try
 {
 // Firefox, Opera 8.0+, Safari
 xmlHttp=new XMLHttpRequest();
 }
catch (e)
 {
 // Internet Explorer
 try
  {
  xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
  }
 catch (e)
  {
  xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
 }
return xmlHttp;
}
```

例子解释：

stateChanged() 和 GetXmlHttpObject 函数与 PHP 和 AJAX 请求 这一节中的例子相同。

showRSS() 函数

每当在下拉框中选择选择时，该函数就会执行：

1. 定义发送到服务器的 url （文件名）
2. 把参数 (q) 添加到 url，参数内容是下拉框中的被选项
3. 添加一个随机数，以防止服务器缓存文件
4. 调用 GetXmlHttpObject 函数来创建 XMLHTTP 对象，并告知该对象在触发一个改变时去执行 stateChanged 函数
5. 通过给定的 url 来打开 XMLHTTP
6. 把 HTTP 请求发动到服务器

PHP 页面

调用 JavaScript 代码的服务器页面是名为 "getrss.php" 的 PHP 文件：

```
<?php
//get the q parameter from URL
$q=$_GET["q"];

//find out which feed was selected
if($q=="Google")
 {
 $xml=("http://news.google.com/news?ned=us&topic=h&output=rss");
 }
elseif($q=="MSNBC")
 {
 $xml=("http://rss.msnbc.msn.com/id/3032091/device/rss/rss.xml");
 }

$xmlDoc = new DOMDocument();
$xmlDoc->load($xml);

//get elements from "<channel>"
$channel=$xmlDoc->getElementsByTagName('channel')->item(0);
$channel_title = $channel->getElementsByTagName('title')
->item(0)->childNodes->item(0)->nodeValue;
$channel_link = $channel->getElementsByTagName('link')
->item(0)->childNodes->item(0)->nodeValue;
$channel_desc = $channel->getElementsByTagName('description')
->item(0)->childNodes->item(0)->nodeValue;

//output elements from "<channel>"
echo("<p><a href='" . $channel_link
 . "'>" . $channel_title . "</a>");
echo("<br />");
echo($channel_desc . "</p>");

//get and output "<item>" elements
$x=$xmlDoc->getElementsByTagName('item');
for ($i=0; $i<=2; $i++)
 {
 $item_title=$x->item($i)->getElementsByTagName('title')
 ->item(0)->childNodes->item(0)->nodeValue;
 $item_link=$x->item($i)->getElementsByTagName('link')
 ->item(0)->childNodes->item(0)->nodeValue;
 $item_desc=$x->item($i)->getElementsByTagName('description')
 ->item(0)->childNodes->item(0)->nodeValue;

 echo ("<p><a href='" . $item_link
 . "'>" . $item_title . "</a>");
 echo ("<br />");
 echo ($item_desc . "</p>");
 }
?>
```

例子解释：

当一个选项从 JavaScript 发送时，会发生：

1. PHP 找出哪个 RSS feed 被选中
2. 为选中的 RSS feed 创建 XML DOM 对象
3. 找到并输出来自 RSS 频道的元素
4. 遍历前三个 RSS 项目中的元素，并进行输出

PHP 和 AJAX 投票

AJAX 投票

在这个 AJAX 实例中，我们将演示一个投票程序，网页在不重新加载的情况下，就可以获得结果。

本例包括四个元素：

- HTML 表单
- JavaScript
- PHP 页面
- 存放结果的文本文件

HTML 表单

这是 HTML 页面。它包含一个简单的 HTML 表单，以及一个与 JavaScript 文件的连接：

```
<html>
<head>
<script src="poll.js"></script>
</head>
<body>

<div id="poll">
<h2>Do you like PHP and AJAX so far?</h2>

<form>
Yes:
<input type="radio" name="vote"
value="0" onclick="getVote(this.value)">
<br />
No:
<input type="radio" name="vote"
value="1" onclick="getVote(this.value)">
</form>
</div>

</body>
</html>
```

例子解释 - HTML 表单

正如您看到的，上面的 HTML 页面包含一个简单的 HTML 表单，其中的 <div> 元素带有两个单选按钮。

表单这样工作：

- 当用户选择 "yes" 或 "no" 时，会触发一个事件

- 当事件触发时，执行 getVote() 函数
- 围绕该表单的是名为 "poll" 的 <div>。当数据从 getVote() 函数返回时，返回的数据会替代该表单。

文本文件

文本文件 (poll_result.txt) 中存储来自投票程序的数据。

它类似这样：

```
0||0
```

第一个数字表示 "Yes" 投票，第二个数字表示 "No" 投票。

注释：记得只允许您的 web 服务器来编辑该文本文件。不要让其他人获得访问权，除了 web 服务器 (PHP)。

JavaScript

JavaScript 代码存储在 "poll.js" 中，并于 HTML 文档相连接：

```
var xmlHttp

function getVote(int)
{
xmlHttp=GetXmlHttpObject()
if (xmlHttp==null)
 {
 alert ("Browser does not support HTTP Request")
 return
 }
var url="poll_vote.php"
url=url+"?vote="+int
url=url+"&sid="+Math.random()
xmlHttp.onreadystatechange=stateChanged
xmlHttp.open("GET",url,true)
xmlHttp.send(null)
}

function stateChanged()
{
 if (xmlHttp.readyState==4 || xmlHttp.readyState=="complete")
 {
 document.getElementById("poll").
 innerHTML=xmlHttp.responseText;
 }
}

function GetXmlHttpObject()
{
var objXMLHttp=null
if (window.XMLHttpRequest)
 {
 objXMLHttp=new XMLHttpRequest()
 }
else if (window.ActiveXObject)
 {
 objXMLHttp=new ActiveXObject("Microsoft.XMLHTTP")
 }
return objXMLHttp
}
```

例子解释：

stateChanged() 和 GetXmlHttpObject 函数与 PHP 和 AJAX 请求 这一节中的例子相同。

getVote() 函数

当用户在 HTML 表单中选择 "yes" 或 "no" 时，该函数就会执行。

1. 定义发送到服务器的 url （文件名）
2. 向 url 添加参数 (vote)，参数中带有输入字段的内容
3. 添加一个随机数，以防止服务器使用缓存的文件
4. 调用 GetXmlHttpObject 函数来创建 XMLHTTP 对象，并告知该对象当触发一个变化时执行 stateChanged 函数
5. 用给定的 url 来打开 XMLHTTP 对象
6. 向服务器发送 HTTP 请求

PHP 页面

由 JavaScript 代码调用的服务器页面是名为 "poll_vote.php" 的一个简单的 PHP 文件。

```
<?php
$vote = $_REQUEST['vote'];

//get content of textfile
$filename = "poll_result.txt";
$content = file($filename);

//put content in array
$array = explode("||", $content[0]);
$yes = $array[0];
$no = $array[1];

if ($vote == 0)
 {
 $yes = $yes + 1;
 }
if ($vote == 1)
 {
 $no = $no + 1;
 }

//insert votes to txt file
$insertvote = $yes."||".$no;
$fp = fopen($filename,"w");
fputs($fp,$insertvote);
fclose($fp);
?>

<h2>Result:</h2>
<table>
<tr>
<td>Yes:</td>
<td>
<img src="poll.gif"
width='<?php echo(100*round($yes/($no+$yes),2)); ?>'
height='20'>
<?php echo(100*round($yes/($no+$yes),2)); ?>%
</td>
</tr>
<tr>
<td>No:</td>
<td>
<img src="poll.gif"
width='<?php echo(100*round($no/($no+$yes),2)); ?>'
height='20'>
<?php echo(100*round($no/($no+$yes),2)); ?>%
</td>
</tr>
</table>
```

例子解释：

所选的值从 JavaScript 传来，然后会发生：

1. 获取 "poll_result.txt" 文件的内容
2. 把文件内容放入变量，并向被选变量累加 1
3. 把结果写入 "poll_result.txt" 文件
4. 输出图形化的投票结果

W3School PHP 参考手册

来源：PHP 参考手册

整理：飞龙

PHP Array 函数

PHP Array 简介

array 函数允许您对数组进行操作。

PHP 支持单维和多维的数组。同时提供了用数据库查询结果来构造数组的函数。

安装

array 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP Array 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
array()	创建数组。	3
array_change_key_case()	返回其键均为大写或小写的数组。	4
array_chunk()	把一个数组分割为新的数组块。	4
array_combine()	通过合并两个数组来创建一个新数组。	5
array_count_values()	用于统计数组中所有值出现的次数。	4
array_diff()	返回两个数组的差集数组。	4
array_diff_assoc()	比较键名和键值，并返回两个数组的差集数组。	4
array_diff_key()	比较键名，并返回两个数组的差集数组。	5
array_diff_uassoc()	通过用户提供的回调函数做索引检查来计算数组的差集。	5
array_diff_ukey()	用回调函数对键名比较计算数组的差集。	5
array_fill()	用给定的值填充数组。	4
array_filter()	用回调函数过滤数组中的元素。	4
array_flip()	交换数组中的键和值。	4
array_intersect()	计算数组的交集。	4
array_intersect_assoc()	比较键名和键值，并返回两个数组的交集数组。	4
array_intersect_key()	使用键名比较计算数组的交集。	5

array_intersect_uassoc()	带索引检查计算数组的交集，用回调函数比较索引。	5
array_intersect_ukey()	用回调函数比较键名来计算数组的交集。	5
array_key_exists()	检查给定的键名或索引是否存在于数组中。	4
array_keys()	返回数组中所有的键名。	4
array_map()	将回调函数作用到给定数组的单元上。	4
array_merge()	把一个或多个数组合并为一个数组。	4
array_merge_recursive()	递归地合并一个或多个数组。	4
array_multisort()	对多个数组或多维数组进行排序。	4
array_pad()	用值将数组填补到指定长度。	4
array_pop()	将数组最后一个单元弹出（出栈）。	4
array_product()	计算数组中所有值的乘积。	5
array_push()	将一个或多个单元（元素）压入数组的末尾（入栈）。	4
array_rand()	从数组中随机选出一个或多个元素，并返回。	4
array_reduce()	用回调函数迭代地将数组简化为单一的值。	4
array_reverse()	将原数组中的元素顺序翻转，创建新的数组并返回。	4
array_search()	在数组中搜索给定的值，如果成功则返回相应的键名。	4
array_shift()	删除数组中的第一个元素，并返回被删除元素的值。	4
array_slice()	在数组中根据条件取出一段值，并返回。	4
array_splice()	把数组中的一部分去掉并用其它值取代。	4
array_sum()	计算数组中所有值的和。	4
array_udiff()	用回调函数比较数据来计算数组的差集。	5
array_udiff_assoc()	带索引检查计算数组的差集，用回调函数比较数据。	5
array_udiff_uassoc()	带索引检查计算数组的差集，用回调函数比较数据和索引。	5
array_uintersect()	计算数组的交集，用回调函数比较数据。	5
array_uintersect_assoc()	带索引检查计算数组的交集，用回调函数比较数据。	5
array_uintersect_uassoc()	带索引检查计算数组的交集，用回调函数比较数据和索引。	5

array_unique()	删除数组中重复的值。	4
array_unshift()	在数组开头插入一个或多个元素。	4
array_values()	返回数组中所有的值。	4
array_walk()	对数组中的每个成员应用用户函数。	3
array_walk_recursive()	对数组中的每个成员递归地应用用户函数。	5
arsort()	对数组进行逆向排序并保持索引关系。	3
asort()	对数组进行排序并保持索引关系。	3
compact()	建立一个数组，包括变量名和它们的值。	4
count()	计算数组中的元素数目或对象中的属性个数。	3
current()	返回数组中的当前元素。	3
each()	返回数组中当前的键／值对并将数组指针向前移动一步。	3
end()	将数组的内部指针指向最后一个元素。	3
extract()	从数组中将变量导入到当前的符号表。	3
in_array()	检查数组中是否存在指定的值。	4
key()	从关联数组中取得键名。	3
krsort()	对数组按照键名逆向排序。	3
ksort()	对数组按照键名排序。	3
list()	把数组中的值赋给一些变量。	3
natcasesort()	用“自然排序”算法对数组进行不区分大小写字母的排序。	4
natsort()	用“自然排序”算法对数组排序。	4
next()	将数组中的内部指针向前移动一位。	3
pos()	current() 的别名。	3
prev()	将数组的内部指针倒回一位。	3
range()	建立一个包含指定范围的元素的数组。	3
reset()	将数组的内部指针指向第一个元素。	3
rsort()	对数组逆向排序。	3
shuffle()	把数组中的元素按随机顺序重新排列。	3
sizeof()	count() 的别名。	3
sort()	对数组排序。	3
uasort()	使用用户自定义的比较函数对数组中的值进行排序并保持索引关联。	3

uksort()	使用用户自定义的比较函数对数组中的键名进行排序。	3
usort()	使用用户自定义的比较函数对数组中的值进行排序。	3

PHP Array 常量

PHP：指示支持该常量的最早的 PHP 版本。

常量	描述	PHP
CASE_LOWER	用在 array_change_key_case() 中将数组键名转换成小写字母。	
CASE_UPPER	用在 array_change_key_case() 中将数组键名转换成大写字母。	
SORT_ASC	用在 array_multisort() 函数中，使其升序排列。	
SORT_DESC	用在 array_multisort() 函数中，使其降序排列。	
SORT_REGULAR	用于对对象进行通常比较。	
SORT_NUMERIC	用于对对象进行数值比较。	
SORT_STRING	用于对对象进行字符串比较。	
SORT_LOCALE_STRING	基于当前区域来对对象进行字符串比较。	4
COUNT_NORMAL		
COUNT_RECURSIVE		
EXTR_OVERWRITE		
EXTR_SKIP		
EXTR_PREFIX_SAME		
EXTR_PREFIX_ALL		
EXTR_PREFIX_INVALID		
EXTR_PREFIX_IF_EXISTS		
EXTR_IF_EXISTS		
EXTR_REFS		

PHP array()

定义和用法

array() 创建数组，带有键和值。如果在规定数组时省略了键，则生成一个整数键，这个 key 从 0 开始，然后以 1 进行递增。

要用 array() 创建一个关联数组，可使用 => 来分隔键和值。

要创建一个空数组，则不传递参数给 array()：

```
$new = array();
```

注意：array() 实际上是一种语言结构 (language construct)，通常用来定义直接量数组，但它的用法和函数的用法很相似，所以我们把它也列到手册中。

语法

```
array(key => value)
```

参数	描述
key	可选。规定 key，类型是数值或字符串。如果未设置，则生成整数类型的 key。
value	必需。规定值。

例子 1

```
<?php
$a=array("a"=>"Dog","b"=>"Cat","c"=>"Horse");
print_r($a);
?>
```

输出：

```
Array ( [a] => Dog [b] => Cat [c] => Horse )
```

例子 2

```
<?php
$a=array("Dog","Cat","Horse");
print_r($a);
?>
```

输出：

```
Array ( [0] => Dog [1] => Cat [2] => Horse )
```

PHP array_change_key_case() 函数

定义和用法

array_change_key_case() 函数将数组的所有的 KEY 都转换为大写或小写。

数组的数字索引不发生变化。如果未提供可选参数（即第二个参数），则默认转换为小写字母。

语法

```
array_change_key_case(array,case)
```

参数	描述
array	必需。规定要使用的数组。
case	可选。可能的值：`CASE_LOWER` - 默认值。以小写字母返回数组的键。`CASE_UPPER` - 以大写字母返回数组的键。

提示和注释：

注释：如果在运行该函数时两个或多个键相同，则最后的元素会覆盖其他元素（参见例子 2）。

例子 1

```
<?php
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
print_r(array_change_key_case($a,CASE_UPPER));
?>
```

输出：

```
Array ( [A] => Cat [B] => Dog [C] => Horse )
```

例子 2

```
<?php
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse","B"=>"Bird");
print_r(array_change_key_case($a,CASE_UPPER));
?>
```

输出：

```
Array ( [A] => Cat [B] => Bird [C] => Horse )
```

PHP array_chunk() 函数

定义和用法

array_chunk() 函数把一个数组分割为新的数组块。

其中每个数组的单元数目由 size 参数决定。最后一个数组的单元数目可能会少几个。

可选参数 preserve_key 是一个布尔值，它指定新数组的元素是否有和原数组相同的键（用于关联数组），还是从 0 开始的新数字键（用于索引数组）。默认是分配新的键。

语法

```
array_chunk(array,size,preserve_key)
```

参数	描述
array	必需。规定要使用的数组。
size	必需。规定每个新数组包含多少个元素。
preserve_key	可选。可能的值：`true` - 保留原始数组中的键名。 `false` - 默认。每个结果数组使用从零开始的新数组索引。

例子 1

```
<?php
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse","d"=>"Cow");
print_r(array_chunk($a,2));
?>
```

输出：

```
Array (
[0] => Array ( [0] => Cat [1] => Dog )
[1] => Array ( [0] => Horse [1] => Cow )
)
```

例子 2

```
<?php
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse","d"=>"Cow");
print_r(array_chunk($a,2,true));
?>
```

输出：

```
Array (
[0] => Array ( [a] => Cat [b] => Dog )
[1] => Array ( [c] => Horse [d] => Cow )
)
```

PHP array_combine() 函数

定义和用法

array_combine() 函数通过合并两个数组来创建一个新数组，其中的一个数组是键名，另一个数组的值为键值。

如果其中一个数组为空，或者两个数组的元素个数不同，则该函数返回 false。

语法

```
array_combine(array1,array2)
```

参数	描述
array1	必需。规定键名。
array2	必需。规定值。

提示和注释

注释：两个参数必须有相同数目的元素。

例子

```
<?php
$a1=array("a","b","c","d");
$a2=array("Cat","Dog","Horse","Cow");
print_r(array_combine($a1,$a2));
?>
```

输出：

```
Array ( [a] => Cat [b] => Dog [c] => Horse [d] => Cow )
```

PHP array_count_values() 函数

定义和用法

array_count_values() 函数用于统计数组中所有值出现的次数。

本函数返回一个数组，其元素的键名是原数组的值，键值是该值在原数组中出现的次数。

语法

```
array_count_values(array)
```

参数	描述
array	必需。规定输入的数组。

例子

```
<?php
$a=array("Cat","Dog","Horse","Dog");
print_r(array_count_values($a));
?>
```

输出：

```
Array ( [Cat] => 1 [Dog] => 2 [Horse] => 1 )
```

PHP array_diff() 函数

定义和用法

array_diff() 函数返回两个数组的差集数组。该数组包括了所有在被比较的数组中，但是不在任何其他参数数组中的键值。

在返回的数组中，键名保持不变。

语法

```
array_diff(array1,array2,array3...)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。

提示和注释

提示：可用一个或任意多个数组与第一个数组进行比较。

注释：仅有值用于比较。

例子

```
<?php
$a1=array(0=>"Cat",1=>"Dog",2=>"Horse");
$a2=array(3=>"Horse",4=>"Dog",5=>"Fish");
print_r(array_diff($a1,$a2));
?>
```

输出：

```
Array ( [0] => Cat )
```

PHP array_diff_assoc() 函数

定义和用法

array_diff_assoc() 函数返回两个数组的差集数组。该数组包括了所有在被比较的数组中，但是不在任何其他参数数组中的键和值。

和 array_diff() 函数 不同，本函数要求键名和键值都进行比较。返回的数组中键名保持不变。

语法

```
array_diff_assoc(array1,array2,array3...)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。

提示和注释

提示：可用一个或任意多个数组与第一个数组进行比较。

注释：键和值都用于比较。

例子

```
<?php
$a1=array(0=>"Cat",1=>"Dog";,2=>"Horse");
$a2=array(0=>"Rat",1=>"Horse";,2=>"Dog");
$a3=array(0=>"Horse",1=>"Dog",2=>"Cat");
print_r(array_diff_assoc($a1,$a2,$a3));
?>
```

输出：

```
Array ( [0] => Cat [2] => Horse )
```

PHP array_diff_key() 函数

定义和用法

array_diff_key() 函数返回一个数组，该数组包括了所有在被比较的数组中，但是不在任何其他参数数组中的键。

语法

```
array_diff_key(array1,array2,array3...)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。

提示和注释

提示：可用一个或任意多个数组与第一个数组进行比较。

注释：仅仅键名用于比较。

例子

```
<?php
$a1=array(0=>"Cat",1=>"Dog",2=>"Horse");
$a2=array(2=>"Bird",3=>"Rat",4=>"Fish");
$a3=array(5=>"Horse",6=>"Dog",7=>"Bird");
print_r(array_diff_key($a1,$a2,$a3));
?>
```

输出：

```
Array ( [0] => Cat [1] => Dog )
```

PHP array_diff_uassoc() 函数

定义和用法

array*diff_uassoc()* 函数使用用户自定义的回调函数 *(callback)* 做索引检查来计算两个或多个数组的差集。返回一个数组，该数组包括了在 _*array1* 中但是不在任何其他参数数组中的值。

注意，与 array_diff() 函数 不同的是，键名也要进行比较。

参数 function 是用户自定义的用来比较两个数组的函数，该函数必须带有两个参数 - 即两个要进行对比的键名。因此与函数 array_diff_assoc() 的行为正好相反，后者是用内部函数进行比较的。

返回的数组中键名保持不变。

语法

```
array_diff_uassoc(array1,array2,array3...,function)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。
function	必需。用户自定义函数的名称。

例子 1

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(3=>"Dog",1=>"Cat",5=>"Horse");
print_r(array_diff_uassoc($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [0] => Dog [2] => Horse )
```

例子 2

如何为该函数分配多个数组：

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(3=>"Dog",1=>"Cat",5=>"Horse");
$a3=array(6=>"Bird",0=>"Dog",5=>"Horse");
print_r(array_diff_uassoc($a1,$a2,$a3,"myfunction"));
?>
```

输出：

```
Array ( [2] => Horse )
```

PHP array_diff_ukey() 函数

定义和用法

array_diff_ukey() 返回一个数组，该数组包括了所有出现在 array1 中但是未出现在任何其它参数数组中的键名的值。注意关联关系保留不变。与 array_diff() 不同的是，比较是根据键名而不是值来进行的。

此比较是通过用户提供的回调函数来进行的。如果认为第一个参数小于，等于，或大于第二个参数时必须分别返回一个小于零，等于零，或大于零的整数。

语法

```
array_diff_ukey(array1,array2,array3...,function)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。
function	必需。用户自定义函数的名称。

例子 1

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(3=>"Rat",1=>"Bird",5=>"Monkey");
print_r(array_diff_ukey($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [0] => Dog [2] => Horse )
```

例子 2

如何为该函数分配多个数组：

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(3=>"Rat",1=>"Bird",5=>"Monkey");
$a3=array(6=>"Dog",7=>"Donkey",0=>"Horse");
print_r(array_diff_ukey($a1,$a2,$a3,"myfunction"));
?>
```

输出：

```
Array ( [2] => Horse )
```

PHP array_fill() 函数

定义和用法

array_fill() 函数用给定的值填充数组，返回的数组有 number 个元素，值为 value。返回的数组使用数字索引，从 start 位置开始并递增。如果 number 为 0 或小于 0，就会出错。

语法

```
array_fill(start,number,value)
```

参数	描述
start	必需。数值，规定键的起始索引。
number	必需。数值，规定填充的数量，其值必须大于 0。
value	必需。规定要插入的值。

例子

```
<?php
$a=array_fill(2,3,"Dog");
print_r($a);
?>
```

输出：

```
Array ( [2] => Dog [3] => Dog [4] => Dog )
```

PHP array_filter() 函数

定义和用法

array_filter() 函数用回调函数过滤数组中的元素，如果自定义过滤函数返回 true，则被操作的数组的当前值就会被包含在返回的结果数组中， 并将结果组成一个新的数组。如果原数组是一个关联数组，键名保持不变。

语法

```
array_filter(array,function)
```

参数	描述
array	必需。规定输入的数组。
function	必需。自定义函数的名称。

例子

```
<?php
function myfunction($v)
{
if ($v==="Horse")
    {
    return true;
    }
return false;
}
$a=array(0=>"Dog",1=>"Cat",2=>"Horse");
print_r(array_filter($a,"myfunction"));
?>
```

输出：

```
Array ( [2] => Horse )
```

PHP array_flip() 函数

定义和用法

array_flip() 函数返回一个反转后的数组，如果同一值出现了多次，则最后一个键名将作为它的值，所有其他的键名都将丢失。

如果原数组中的值的数据类型不是字符串或整数，函数将报错。

语法

```
array_flip(array)
```

参数	描述
array	必需。规定输入的数组。

例子

```
<?php
$a=array(0=>"Dog",1=>"Cat",2=>"Horse");print_r(array_flip($a));
?>
```

输出：

```
Array ( [Dog] => 0 [Cat] => 1 [Horse] => 2 )
```

PHP array_intersect() 函数

定义和用法

array_intersect() 函数返回两个或多个数组的交集数组。

结果数组包含了所有在被比较数组中，也同时出现在所有其他参数数组中的值，键名保留不变。

注释：仅有值用于比较。

语法

```
array_intersect(array1,array2,array3...)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。

例子

```
<?php
$a1=array(0=>"Cat",1=>"Dog",2=>"Horse");
$a2=array(3=>"Horse",4=>"Dog",5=>"Fish");
print_r(array_intersect($a1,$a2));
?>
```

输出：

```
Array ( [1] => Dog [2] => Horse )
```

PHP array_intersect_assoc() 函数

定义和用法

array_intersect_assoc() 函数返回两个或多个数组的交集数组。

与 array_intersect() 函数 不同的是，本函数除了比较键值，还比较键名。返回的数组中元素的键名保持不变。

语法

```
array_intersect_assoc(array1,array2,array3...)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。

例子 1

```
<?php
$a1=array(0=>"Cat",1=>"Dog",2=>"Horse");
$a2=array(3=>"Horse",1=>"Dog",0=>"Cat");
print_r(array_intersect_assoc($a1,$a2));
?>
```

输出：

```
Array ( [0] => Cat [1] => Dog )
```

例子 2

```
<?php
$a1=array(0=>"Cat",1=>"Dog",2=>"Horse");
$a2=array(3=>"Horse",1=>"Dog",5=>"Fish");
$a3=array(6=>"Cow",1=>"Dog",8=>"Fish");
print_r(array_intersect_assoc($a1,$a2,$a3));
?>
```

输出：

```
Array ( [1] => Dog )
```

PHP array_intersect_key() 函数

定义和用法

array_intersect_key() 函数使用键名比较计算数组的交集。

array_intersect_key() 返回一个数组，该数组包含了所有出现在被比较的数组中并同时出现在所有其它参数数组中的键名的值。

注释：仅有键名用于比较。

语法

```
array_intersect_key(array1,array2,array3...)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。

例子 1

```
<?php
$a1=array(0=>"Cat",1=>"Dog",2=>"Horse");
$a2=array(2=>"Bird",0=>"Cat",4=>"Fish");
print_r(array_intersect_key($a1,$a2));
?>
```

输出：

```
Array ( [0] => Cat [2] => Horse )
```

例子 2

```
<?php
$a1=array(0=>"Cat",1=>"Dog",2=>"Horse");
$a2=array(2=>"Bird",3=>"Rat",4=>"Fish");
$a3=array(2=>"Dog",6=>"Cow",7=>"Bird");
print_r(array_intersect_key($a1,$a2,$a3));
?>
```

输出：

```
Array ( [2] => Horse )
```

PHP array_intersect_uassoc() 函数

定义和用法

array_intersect_uassoc() 函数使用用户自定义的回调函数计算数组的交集，用回调函数比较索引。

array*intersect_uassoc()* 返回一个数组，该数组包含了所有在 _*array1* 中也同时出现在所有其它参数数组中的值。返回的数组中键名保持不变。

注意，与 array_intersect() 不同的是除了比较键值，还要比较键名。

此比较是通过用户提供的回调函数来进行的。该函数带有两个参数，即两个要进行对比的键名。如果第一个参数小于第二个参数，则函数要返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个参数，则返回一个正数。

语法

```
array_intersect_uassoc(array1,array2,array3...,function)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。
function	用户自定义函数的名称。

例子 1

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(3=>"Dog",1=>"Cat",5=>"Horse");
print_r(array_intersect_uassoc($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [1] => Cat )
```

例子 2

如何为函数分配多个数组：

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(0=>"Dog",1=>"Cat",5=>"Horse");
$a3=array(6=>"Bird",0=>"Dog",5=>"Horse");
print_r(array_intersect_uassoc($a1,$a2,$a3,"myfunction"));
?>
```

输出：

```
Array ( [0] => Dog )
```

PHP array_intersect_ukey() 函数

定义和用法

array_intersect_ukey() 函数用回调函数比较键名来计算数组的交集。

array_intersect_ukey() 返回一个数组，该数组包含了所有出现在 array1 中并同时出现在所有其它参数数组中的键名的值。

此比较是通过用户提供的回调函数来进行的。该函数带有两个参数，即两个要进行对比的键名。如果第一个参数小于第二个参数，则函数要返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个参数，则返回一个正数。

语法

```
array_intersect_uassoc(array1,array2,array3...,function)
```

参数	描述
array1	必需。与其他数组进行比较的第一个数组。
array2	必需。与第一个数组进行比较的数组。
array3	可选。与第一个数组进行比较的数组。可以有多个。
function	用户自定义函数的名称。

例子 1

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(3=>"Rat",1=>"Bird",5=>"Monkey");
print_r(array_intersect_ukey($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [1] => Cat )
```

例子 2

如何为函数分配多个数组：

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
if ($v1>$v2)
    {
    return 1;
    }
else
    {
    return -1;
    }
}
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse");
$a2=array(0=>"Rat",1=>"Bird",5=>"Monkey");
$a3=array(6=>"Dog",7=>"Donkey",0=>"Horse");
print_r(array_intersect_ukey($a1,$a2,$a3,"myfunction"));
?>
```

输出：

```
Array ( [0] => Dog )
```

PHP array_key_exists() 函数

定义和用法

array_key_exists() 函数判断某个数组中是否存在指定的 key，如果该 key 存在，则返回 true，否则返回 false。

语法

```
array_key_exists(key,array)
```

参数	描述
key	必需。规定键名。
array	必需。规定输入的数组。

例子 1

```
<?php
$a=array("a"=>"Dog","b"=>"Cat");
if (array_key_exists("a",$a))
    {
    echo "Key exists!";
    }
else
    {
    echo "Key does not exist!";
    }
?>
```

输出：

```
Key exists!
```

例子 2

```
<?php
$a=array("a"=>"Dog","b"=>"Cat");
if (array_key_exists("c",$a))
    {
    echo "Key exists!";
    }
else
    {
    echo "Key does not exist!";
    }
?>
```

输出：

```
Key does not exist!
```

例子 2

```
<?php
$a=array("Dog",Cat");
if (array_key_exists(0,$a))
    {
    echo "Key exists!";
    }
else
    {
    echo "Key does not exist!";
    }
?>
```

输出：

```
Key exists!
```

PHP array_keys() 函数

定义和用法

array_keys() 函数返回包含数组中所有键名的一个新数组。

如果提供了第二个参数，则只返回键值为该值的键名。

如果 strict 参数指定为 true，则 PHP 会使用全等比较 (===) 来严格检查键值的数据类型。

语法

```
array_keys(array,value)
```

参数	描述
array	必需。规定输入的数组。
value	可选。指定值的索引（键）。
strict	可选。与 value 参数一起使用。可能的值：`true` - 根据类型返回带有指定值的键名。`false` - 默认值。不依赖类型。

例子 1

```
<?php
$a=array("a"=>"Horse","b"=>"Cat","c"=>"Dog");
print_r(array_keys($a));
?>
```

输出：

```
Array ( [0] => a [1] => b [2] => c )
```

例子 2

使用 value 参数：

```
<?php
$a=array("a"=>"Horse","b"=>"Cat","c"=>"Dog");
print_r(array_keys($a,"Dog"));
?>
```

输出：

```
Array ( [0] => c)
```

例子 3

使用 strict 参数 (false)：

```
<?php
$a=array(10,20,30,"10");
print_r(array_keys($a,"10",false));
?>
```

输出：

```
Array ( [0] => 0 [1] => 3 )
```

例子 4

使用 strict 参数 (true)：

```
<?php
$a=array(10,20,30,"10");
print_r(array_keys($a,"10",true));
?>
```

输出：

```
Array ( [0] => 3)
```

PHP array_map() 函数

定义和用法

array_map() 函数返回用户自定义函数作用后的数组。回调函数接受的参数数目应该和传递给 array_map() 函数的数组数目一致。

语法

```
array_map(function,array1,array2,array3...)
```

参数	描述
function	必需。用户自定义函数的名称，或者是 null。
array1	必需。规定数组。
array2	可选。规定数组。
array3	可选。规定数组。

例子 1

```
<?php
function myfunction($v)
{
if ($v==="Dog")
    {
    return "Fido";
    }
return $v;
}
$a=array("Horse","Dog","Cat");
print_r(array_map("myfunction",$a));
?>
```

输出：

```
Array ( [0] => Horse [1] => Fido [2] => Cat )
```

例子 2

使用多个参数：

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return "same";
    }
return "different";
}
$a1=array("Horse","Dog","Cat");
$a2=array("Cow","Dog","Rat");
print_r(array_map("myfunction",$a1,$a2));
?>
```

输出：

```
Array ( [0] => different [1] => same [2] => different )
```

例子 3

请看当自定义函数名设置为 null 时的情况：

```
<?php
$a1=array("Dog","Cat");
$a2=array("Puppy","Kitten");
print_r(array_map(null,$a1,$a2));
?>
```

输出：

```
Array (
[0] => Array ( [0] => Dog [1] => Puppy )
[1] => Array ( [0] => Cat [1] => Kitten )
)
```

PHP array_merge() 函数

定义和用法

array_merge() 函数把两个或多个数组合并为一个数组。

如果键名有重复，该键的键值为最后一个键名对应的值（后面的覆盖前面的）。如果数组是数字索引的，则键名会以连续方式重新索引。

注释：如果仅仅向 array_merge() 函数输入了一个数组，且键名是整数，则该函数将返回带有整数键名的新数组，其键名以 0 开始进行重新索引。（参见例子 2）

语法

```
array_merge(array1,array2,array3...)
```

参数	描述
array1	必需。输入的第一个数组。
array2	必需。输入的第二个数组。
array3	可选。可指定的多个输入数组。

例子 1

```
<?php
$a1=array("a"=>"Horse","b"=>"Dog");
$a2=array("c"=>"Cow","b"=>"Cat");
print_r(array_merge($a1,$a2));
?>
```

输出：

```
Array ( [a] => Horse [b] => Cat [c] => Cow )
```

例子 2

仅使用一个数组参数：

```
<?php
$a=array(3=>"Horse",4=>"Dog");
print_r(array_merge($a));
?>
```

输出：

```
Array ( [0] => Horse [1] => Dog )
```

PHP array_merge_recursive() 函数

定义和用法

array_merge_recursive() 函数与 array_merge() 函数 一样，将一个或多个数组的元素的合并起来，一个数组中的值附加在前一个数组的后面。并返回作为结果的数组。

但是，与 array_merge() 不同的是，当有重复的键名时，值不会被覆盖，而是将多个相同键名的值递归组成一个数组。（参见例子 1）

语法

```
array_merge_recursive(array1,array2,array3...)
```

参数	描述
array1	必需。输入的第一个数组。
array2	必需。输入的第二个数组。
array3	可选。可指定的多个输入数组。

提示和注释

注释：当向 array_merge_recursive() 函数仅仅输入一个数组时，结果与 array_merge() 相同。

例子 1

```
<?php
$a1=array("a"=>"Horse","b"=>"Dog");
$a2=array("c"=>"Cow","b"=>"Cat");
print_r(array_merge_recursive($a1,$a2));
?>
```

输出：

```
Array (
[a] => Horse
[b] => Array ( [0] => Dog [1] => Cat )
[c] => Cow
)
```

PHP array_multisort() 函数

定义和用法

array_multisort() 函数对多个数组或多维数组进行排序。

参数中的数组被当成一个表的列并以行来进行排序 - 这类似 SQL 的 ORDER BY 子句的功能。第一个数组是要排序的主要数组。数组中的行（值）比较为相同的话，就会按照下一个输入数组中相应值的大小进行排序，依此类推。

第一个参数是数组，随后的每一个参数可能是数组，也可能是下面的排序顺序标志（排序标志用于更改默认的排列顺序）之一：

- SORT_ASC - 默认，按升序排列。(A-Z)
- SORT_DESC - 按降序排列。(Z-A)

随后可以指定排序的类型：

- SORT_REGULAR - 默认。将每一项按常规顺序排列。
- SORT_NUMERIC - 将每一项按数字顺序排列。
- SORT_STRING - 将每一项按字母顺序排列。

语法

```
array_multisort(array1,sorting order,sorting type,array2,array3...)
```

参数	描述
array1	必需。规定输入的数组。
sorting order	可选。规定排列顺序。可能的值是 SORT_ASC 和 SORT_DESC。
sorting type	可选。规定排序类型。可能的值是SORT_REGULAR、SORT_NUMERIC和SORT_STRING。
array2	可选。规定输入的数组。
array3	可选。规定输入的数组。

提示和注释

注释：字符串键名将被保留，但是数字键将被重新索引，从 0 开始，并以 1 递增。

注释：您可以在每个数组后设置排序顺序和排序类型。如果没有设置，每个数组参数会使用默认值。

例子 1

```
<?php
$a1=array("Dog","Cat");
$a2=array("Fido","Missy");
array_multisort($a1,$a2);
print_r($a1);
print_r($a2);
?>
```

输出：

```
Array ( [0] => Cat [1] => Dog )
Array ( [0] => Missy [1] => Fido )
```

例子 2

当两个值相同时如何排序：

```
<?php
$a1=array("Dog","Dog","Cat");
$a2=array("Pluto","Fido","Missy");
array_multisort($a1,$a2);
print_r($a1);
print_r($a2);
?>
```

输出：

```
Array ( [0] => Cat [1] => Dog [2] => Dog )
Array ( [0] => Missy [1] => Fido [2] => Pluto )
```

例子 3

带有排序参数：

```
<?php
$a1=array("Dog","Dog","Cat");
$a2=array("Pluto","Fido","Missy");
array_multisort($a1,SORT_ASC,$a2,SORT_DESC);
print_r($a1);
print_r($a2);
?>
```

输出：

```
Array ( [0] => Cat [1] => Dog [2] => Dog )
Array ( [0] => Missy [1] => Pluto [2] => Fido )
```

PHP array_pad() 函数

定义和用法

array_pad() 函数向一个数组插入带有指定值的指定数量的元素。

语法

```
array_pad(array,size,value)
```

参数	描述
array	必需。规定数组。
size	必需。指定的长度。整数则填补到右侧，负数则填补到左侧。
value	必需。用来填补的值。

提示和注释

提示：如何设置了负的长度值，该函数会在原始数组之前插入新的元素。（参见例子 2）

注释：如果 size 参数小于原始数组的长度，该函数不会删除任何元素。

例子 1

```
<?php
$a=array("Dog","Cat");
print_r(array_pad($a,5,0));
?>
```

输出：

```
Array ( [0] => Dog [1] => Cat [2] => 0 [3] => 0 [4] => 0 )
```

例子 2

带有负的 size 参数：

```
<?php
$a=array("Dog","Cat");
print_r(array_pad($a,-5,0));
?>
```

输出：

```
Array ( [0] => 0 [1] => 0 [2] => 0 [3] => Dog [4] => Cat )
```

PHP array_pop() 函数

定义和用法

array_pop() 函数删除数组中的最后一个元素。

语法

```
array_pop(array)
```

参数	描述
array	必需。规定输入的数组参数。

例子

```
<?php
$a=array("Dog","Cat","Horse");
array_pop($a);
print_r($a);
?>
```

输出：

```
Array ( [0] => Dog [1] => Cat )
```

PHP array_product() 函数

定义和用法

array_product() 函数计算并返回数组中所有值的乘积。

语法

```
array_product(array)
```

参数	描述
array	必需。规定输入的数组参数。

例子

```
<?php
$a=array(5,5);
echo(array_product($a));
?>
```

输出：

```
25
```

PHP array_push() 函数

定义和用法

array_push() 函数向第一个参数的数组尾部添加一个或多个元素（入栈），然后返回新数组的长度。

该函数等于多次调用 $array[] = $value。

语法

```
array_push(array,value1,value2...)
```

参数	描述
array	必需。规定一个数组。
value1	必需。规定要添加的值。
value2	可选。规定要添加的值。

提示和注释

注释：即使数组中有字符串键名，您添加的元素也始终是数字键。（参见例子 2）

注释：如果用 array*push()* 来给数组增加一个单元，还不如用 $array[] =_，因为这样没有调用函数的额外负担。

注释：如果第一个参数不是数组，array_push() 将发出一条警告。这和 $var[] 的行为不同，后者会新建一个数组。

例子 1

```
<?php
$a=array("Dog","Cat");
array_push($a,"Horse","Bird");
print_r($a);
?>
```

输出：

```
Array ( [0] => Dog [1] => Cat [2] => Horse [3] => Bird )
```

例子 2

带有字符串键的数组：

```
<?php
$a=array("a"=>"Dog","b"=>"Cat");
array_push($a,"Horse","Bird");
print_r($a);
?>
```

输出：

```
Array ( [a] => Dog [b] => Cat [0] => Horse [1] => Bird )
```

PHP array_rand() 函数

定义和用法

array_rand() 函数从数组中随机选出一个或多个元素，并返回。

第二个参数用来确定要选出几个元素。如果选出的元素不止一个，则返回包含随机键名的数组，否则返回该元素的键名。

注释：自 PHP 4.2.0 起，不再需要用 srand() 或 mt_srand() 函数给随机数发生器播种，现已被自动完成。

语法

```
array_rand(array,number)
```

参数	描述
array	必需。规定输入的数组参数。
number	可选。默认是 1。规定返回多少个随机的元素。

例子 1

```
<?php
$a=array("a"=>"Dog","b"=>"Cat","c"=>"Horse");
print_r(array_rand($a,1));
?>
```

输出：

```
b
```

例子 2

带有字符串键的数组：

```
<?php
$a=array("a"=>"Dog","b"=>"Cat","c"=>"Horse");
print_r(array_rand($a,2));
?>
```

输出：

```
Array ( [0] => c [1] => b )
```

PHP array_reduce() 函数

定义和用法

array_reduce() 函数用回调函数迭代地将数组简化为单一的值。如果指定第三个参数，则该参数将被当成是数组中的第一个值来处理，或者如果数组为空的话就作为最终返回值。

语法

```
array_reduce(array,function,initial)
```

参数	描述
array	必需。规定输入的数组。
function	必需。规定自定义回调函数的名称。
initial	可选。如果规定了该参数，该参数将作为数组中的第一个值来处理。

例子 1

```
<?php
function myfunction($v1,$v2)
{
return $v1 . "-" . $v2;
}
$a=array("Dog","Cat","Horse");
print_r(array_reduce($a,"myfunction"));
?>
```

输出：

```
-Dog-Cat-Horse
```

例子 2

带有 initial 参数：

```
<?php
function myfunction($v1,$v2)
{
return $v1 . "-" . $v2;
}
$a=array("Dog","Cat","Horse");
print_r(array_reduce($a,"myfunction",5));
?>
```

输出：

```
5-Dog-Cat-Horse
```

例子 3

返回总和：

```
<?php
function myfunction($v1,$v2)
{
return $v1+$v2;
}
$a=array(10,15,20);
print_r(array_reduce($a,"myfunction",5));
?>
```

输出：

```
50
```

PHP array_reverse() 函数

定义和用法

array_reverse() 函数将原数组中的元素顺序翻转，创建新的数组并返回。如果第二个参数指定为 true，则元素的键名保持不变，否则键名将丢失。

语法

```
array_reverse(array,preserve)
```

参数	描述
array	必需。规定数组。
preserve	可选。规定是否保留原始数组的键名。可能的值：true 和 false。这个参数是 PHP 4.0.3 中新加的。

例子

```
<?php
$a=array("a"=>"Dog","b"=>"Cat","c"=>"Horse");
print_r(array_reverse($a));
?>
```

输出：

```
Array ( [c] => Horse [b] => Cat [a] => Dog )
```

PHP array_search() 函数

定义和用法

array_search() 函数与 in_array() 一样，在数组中查找一个键值。如果找到了该值，匹配元素的键名会被返回。如果没找到，则返回 false。

在 PHP 4.2.0 之前，函数在失败时返回 null 而不是 false。

如果第三个参数 strict 被指定为 true，则只有在数据类型和值都一致时才返回相应元素的键名。

语法

```
array_search(value,array,strict)
```

参数	描述
value	必需。规定在数组中搜索的值。
array	必需。被搜索的数组。
strict	可选。可能的值： `true` `false` - 默认 如果值设置为 true，还将在数组中检查给定值的类型。（参见例子 2）

例子 1

```
<?php
$a=array("a"=>"Dog","b"=>"Cat","c"=>"Horse");
echo array_search("Dog",$a);
?>
```

输出：

```
a
```

例子 2

```
<?php
$a=array("a"=>"5","b"=>5,"c"=>"5");
echo array_search(5,$a,true);
?>
```

输出：

```
b
```

PHP array_shift() 函数

定义和用法

array_shift() 函数删除数组中的第一个元素，并返回被删除元素的值。

注释：如果键是数字的，所有元素都将获得新的键，从 0 开始，并以 1 递增。（参见例子 2）。

语法

```
array_shift(array)
```

参数	描述
array	必需。规定输入的数组。

例子 1

```
<?php
$a=array("a"=>"Dog","b"=>"Cat","c"=>"Horse");
echo array_shift($a);
print_r ($a);
?>
```

输出：

```
Dog
Array ( [b] => Cat [c] => Horse )
```

例子 2

带有数字键：

```
<?php
$a=array(0=>"Dog",1=>"Cat",2=>"Horse");
echo array_shift($a);
print_r ($a);
?>
```

输出：

```
Dog
Array ( [0] => Cat [1] => Horse )
```

PHP array_slice() 函数

定义和用法

array_slice() 函数在数组中根据条件取出一段值，并返回。

注释：如果数组有字符串键，所返回的数组将保留键名。（参见例子 4）

语法

```
array_slice(_array_,_offset_,_length_,_preserve_)
```

参数	描述
array	必需。规定输入的数组。
offset	必需。数值。规定取出元素的开始位置。如果是正数，则从前往后开始取，如果是负值，从后向前取 *offset* 绝对值。
length	可选。数值。规定被返回数组的长度。如果 *length* 为正，则返回该数量的元素。如果 *length* 为负，则序列将终止在距离数组末端这么远的地方。如果省略，则序列将从 *offset* 开始直到 *array* 的末端。
preserve	可选。可能的值： `true` - 保留键 `false` - 默认 - 重置键

例子 1

```
<?php
$a=array(0=>"Dog",1=>"Cat",2=>"Horse",3=>"Bird");
print_r(array_slice($a,1,2));
?>
```

输出：

```
Array ( [0] => Cat [1] => Horse )
```

例子 2

带有负的 offset 参数：

```
<?php
$a=array(0=>"Dog",1=>"Cat",2=>"Horse",3=>"Bird");
print_r(array_slice($a,-2,1));
?>
```

输出：

```
Array ( [0] => Horse )
```

例子 3

preserve 参数设置为 true：

```
<?php
$a=array(0=>"Dog",1=>"Cat",2=>"Horse",3=>"Bird");
print_r(array_slice($a,1,2,true));
?>
```

输出：

```
Array ( [1] => Cat [2] => Horse )
```

例子 4

带有字符串键：

```
<?php
$a=array("a"=>"Dog","b"=>"Cat","c"=>"Horse","d"=>"Bird");
print_r(array_slice($a,1,2));
?>
```

输出：

```
Array ( [b] => Cat [c] => Horse )
```

PHP array_splice() 函数

定义和用法

array_splice() 函数与 array_slice() 函数类似，选择数组中的一系列元素，但不返回，而是删除它们并用其它值代替。

如果提供了第四个参数，则之前选中的那些元素将被第四个参数指定的数组取代。

最后生成的数组将会返回。

语法

```
array_splice(array,offset,length,array)
```

参数	描述
array	必需。规定数组。
offset	必需。数值。如果 offset 为正，则从输入数组中该值指定的偏移量开始移除。如果 offset 为负，则从输入数组末尾倒数该值指定的偏移量开始移除。
length	可选。数值。如果省略该参数，则移除数组中从 offset 到 结尾的所有部分。如果指定了 length 并且为正值，则移除这么多元素。如果指定了 length 且为负值，则移除从 offset 到数组末尾倒数 length 为止中间所有的元素。
array	被移除的元素由此数组中的元素替代。如果没有移除任何值，则此数组中的元素将插入到指定位置。

提示和注释

提示：如果函数没有删除任何元素 (length=0)，则替代数组将从 start 参数的位置插入。（参见例子 3）

注释：不保留替代数组中的键。

例子 1

```
<?php
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse",3=>"Bird");
$a2=array(0=>"Tiger",1=>"Lion");
array_splice($a1,0,2,$a2);
print_r($a1);
?>
```

输出：

```
Array ( [0] => Tiger [1] => Lion [2] => Horse [3] => Bird )
```

例子 2

与例子 1 相同，但是输出返回的数组：

```
<?php
$a1=array(0=>"Dog",1=>"Cat",2=>"Horse",3=>"Bird");
$a2=array(0=>"Tiger",1=>"Lion");
print_r(array_splice($a1,0,2,$a2));
?>
```

输出：

```
Array ( [0] => Dog [1] => Cat )
```

例子 3

length 参数设置为 0：

```
<?php
$a1=array(0=>"Dog",1=>"Cat");
$a2=array(0=>"Tiger",1=>"Lion");
array_splice($a1,1,0,$a2);
print_r($a1);
?>
```

输出：

```
Array ( [0] => Dog [1] => Tiger [2] => Lion [3] => Cat )
```

PHP array_sum() 函数

定义和用法

array_sum() 函数返回数组中所有值的总和。

如果所有值都是整数，则返回一个整数值。如果其中有一个或多个值是浮点数，则返回浮点数。

PHP 4.2.1 之前的版本修改了传入的数组本身，将其中的字符串值转换成数值（大多数情况下都转换成了零，根据具体制而定）。

语法

```
array_sum(array)
```

参数	描述
array	必需。规定输入的数组。

例子

```
<?php
$a=array(0=>"5",1=>"15",2=>"25");
echo array_sum($a);
?>
```

输出：

```
45
```

PHP array_udiff() 函数

定义和用法

array_udiff() 函数返回一个数组，该数组包括了所有在被比较数组中，但是不在任何其它参数数组中的值，键名保留不变。

array_udiff() 函数与 array_diff() 函数 的行为不同，后者用内部函数进行比较。

数据的比较是用 array*udiff()* 函数的 _function 进行的。*function* 函数带有两个将进行比较的参数。如果第一个参数小于第二个参数，则函数返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个，则返回一个正数。

语法

```
array_udiff(_array1_,_array2_,_array3_...,_function_)
```

参数	描述
array1	必需。被比较的数组。
array2	必需。用来做比较的数组。
array3	可选。用来做比较的数组，可有多个。
function	必需。自定义的比较回调函数。

注释和提示

注释：该函数只进行键值的比较，不比较键名。如 "a"=>1 和 "b"=>1 这两个元素视作相等的。

注释：array_udiff() 函数仅检查多维数组中的一维。

例子

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
return 1;
}
$a1=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
$a2=array(1=>"Cat",2=>"Dog",3=>"Fish");
print_r(array_udiff($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [c] => Horse )
```

PHP array_udiff_assoc() 函数

定义和用法

array*udiff_assoc()* 函数返回 *_array1* 中存在但其它数组中都不存在的部分。

注意与 array_diff() 以及 array_udiff() 不同的是键名也用于比较。同时进行键名和键值的比较。如 "a"=>1 和 "b"=>1 这两个元素是不相等的。

数组数据的比较是用用户提供的回调函数进行的。在此方面和 array_diff_assoc() 的行为正好相反，后者是用内部函数进行比较的。

array*udiff_assoc()* 函数的 *_function* 参数指定的函数用于比较元素是否相等。*function* 函数带有两个将进行比较的参数。如果第一个参数小于第二个参数，则函数返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个，则返回一个正数。

语法

```
array_udiff_assoc(array1,array2,array3...,function)
```

参数	描述
array1	必需。被比较的数组。
array2	必需。用来做比较的数组。
array3	可选。用来做比较的数组，可有多个。
function	可选。自定义的比较回调函数。

例子

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
return 1;
}
$a1=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
$a2=array("a"=>"Cat","b"=>"Horse","c"=>"Dog");
print_r(array_udiff_assoc($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [b] => Dog [c] => Horse )
```

PHP array_udiff_uassoc() 函数

定义和用法

array*udiff_uassoc()* 函数返回 *_array1* 数组中存在但其它数组中都不存在的部分。返回的数组中键名保持不变。

注意与 array_diff() 以及 array_udiff() 不同的是键名也用于比较。同时进行键名和键值的比较，如 "a"=>1 和 "b"=>1 这两个元素是不相等的。

对键名（索引）的检查也是由回调函数 *function1* 进行的。这和 array_udiff_assoc() 的行为不同，后者是用内部函数比较索引的。

数组数据的比较是使用用户提供的回调函数 *function2* 进行的。在此方面和 array_diff_assoc() 的行为正好相反，后者是用内部函数进行比较的。

这两个函数都带有两个将进行比较的参数。如果第一个参数小于第二个参数，则函数返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个，则返回一个正数。

语法

```
array_udiff_uassoc(array1,array2,array3...,function1,function2)
```

参数	描述
array1	必需。被比较的数组。
array2	必需。用来做比较的数组。
array3	可选。用来做比较的数组，可有多个。
function1	必需。比较键名的自定义函数。
function2	必需。比较值的自定义函数。

注释：*function1* 指定的函数用于比较键名是否相等。*function2* 指定的函数用于比较键值是否相等。

例子

```
<?php
function myfunction_key($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
return 1;
}
function myfunction_value($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
return 1;
}
$a1=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
$a2=array("a"=>"Cat","b"=>"Dog","c"=>"Fish");
print_r(array_udiff_uassoc($a1,$a2,"myfunction_key","myfunction_value"));
?>
```

输出：

```
Array ( [c] => Horse )
```

PHP array_uintersect() 函数

定义和用法

array_uintersect() 函数计算数组的交集，用回调函数比较数据。

array*uintersect()* 返回一个数组，该数组包含了所有在 *_array1* 中也同时出现在所有其它参数数组中的值。数据（键值）比较是用回调函数进行的。

语法

```
array_uintersect(array1,array2,array3...,function)
```

参数	描述
array1	必需。被比较的数组。
array2	必需。用来做比较的数组。
array3	可选。用来做比较的数组，可有多个。
function	必需。自定义函数的名称。

说明

使用用户自定义的回调函数 *function* 来计算两个或多个数组的交集（即在 *array1* 中存在同时也在其它任何数组中存在的所有数组元素），并返回结果数组。

只进行键值的比较，不比较键名，如 "a"=>1 和 "b"=>1 这两个元素视作相等的。

function 参数指定的函数用于比较元素是否相等。function 函数带有两个将进行比较的参数。如果第一个参数小于第二个参数，则函数返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个，则返回一个正数。

返回的数组中键名保持不变。

例子

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
  {
  return 0;
  }
if ($v1 > $v2) return 1;
  {
  return -1;
  }
return 1;
}
$a1=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
$a2=array(1=>"Cat",2=>"Dog",3=>"Fish");
print_r(array_uintersect($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [a] => Cat [b] => Dog )
```

PHP array_uintersect_assoc() 函数

定义和用法

array_uintersect_assoc() 函数带索引检查计算数组的交集，用回调函数比较数据。

array*uintersect_assoc()* 返回一个数组，该数组包含了所有在 _*array1* 中也同时出现在所有其它参数数组中的值。

注意，与 array_uintersect() 不同的是键名也要比较。数据（键值）是用回调函数比较的。

语法

```
array_uintersect_assoc(array1,array2,array3...,function)
```

参数	描述
array1	必需。被比较的数组。
array2	必需。用来做比较的数组。
array3	可选。用来做比较的数组，可有多个。
function	必需。自定义函数的名称。

说明

使用用户自定义的回调函数 *function* 来计算两个或多个数组的交集（即在 *array1* 中存在，同时也在其它任何数组中存在的所有数组元素），并返回结果数组。

同时进行键名和键值的比较，如 "a"=>1 和 "b"=>1 这两个元素是不相等的。

function 参数指定的函数用于比较元素是否相等。*function* 函数带有两个将进行比较的参数。如果第一个参数小于第二个参数，则函数返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个，则返回一个正数。

返回的数组中键名保持不变。

例子

```
<?php
function myfunction($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
return 1;
}
$a1=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
$a2=array("a"=>"Cat","b"=>"Horse","c"=>"Dog");
print_r(array_uintersect_assoc($a1,$a2,"myfunction"));
?>
```

输出：

```
Array ( [a] => Cat )
```

PHP array_uintersect_uassoc() 函数

定义和用法

array_uintersect_uassoc 函数带索引检查计算数组的交集，用回调函数来比较数据和索引。

array*uintersect_uassoc()* 返回一个数组，该数组包含了所有在 _*array1* 中也同时出现在所有其它参数数组中的值。

注意，与 array_uintersect() 不同的是键名也要比较。键值和键名（索引）都是用回调函数比较的。

语法

```
array_uintersect_uassoc(array1,array2,array3...,function1,function2)
```

参数	描述
array1	必需。被比较的数组。
array2	必需。用来做比较的数组。
array3	可选。用来做比较的数组，可有多个。
function1	必需。比较键名的自定义函数。
function2	必需。比较键值的自定义函数。

说明

使用用户自定义的回调函数 *function1* 和 *function2* 来计算两个或多个数组的交集（即在 *array1* 中存在，同时也在其它任何数组中存在的所有数组元素），并返回结果数组。

同时进行键名和键值的比较，如 "a"=>1 和 "b"=>1 这两个元素是不相等的。

function1 指定的函数用于比较键名是否相等。*function2* 指定的函数用于比较键值是否相等。这两个函数都带有两个将进行比较的参数。如果第一个参数小于第二个参数，则函数返回一个负数，如果两个参数相等，则要返回 0，如果第一个参数大于第二个，则返回一个正数。

返回的数组中键名保持不变。

例子

```
<?php
function myfunction_key($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
return 1;
}

function myfunction_value($v1,$v2)
{
if ($v1===$v2)
    {
    return 0;
    }
return 1;
}
$a1=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
$a2=array("a"=>"Cat","b"=>"Dog","c"=>"Dog");
print_r(array_uintersect_uassoc($a1,$a2,"myfunction_key","myfunction_value"));
?>
```

输出：

```
Array ( [a] => Cat [b] => Dog )
```

PHP array_unique() 函数

定义和用法

array_unique() 函数移除数组中的重复的值，并返回结果数组。

当几个数组元素的值相等时，只保留第一个元素，其他的元素被删除。

返回的数组中键名不变。

语法

```
array_unique(array)
```

参数	描述
array	必需。规定输入的数组。

说明

array*unique()* 先将值作为字符串排序，然后对每个值只保留第一个遇到的键名，接着忽略所有后面的键名。这并不意味着在未排序的 _*array* 中同一个值的第一个出现的键名会被保留。

提示和注释

注释：被返回的数组将保持第一个数组元素的键类型。

例子

```
<?php
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Cat");
print_r(array_unique($a));
?>
```

输出：

```
Array ( [a] => Cat [b] => Dog )
```

PHP array_unshift() 函数

定义和用法

array_unshift() 函数在数组开头插入一个或多个元素。

被加上的元素作为一个整体添加，这些元素在数组中的顺序和在参数中的顺序一样。

该函数会返回数组中元素的个数。

语法

```
array_unshift(array,value1,value2,value3...)
```

参数	描述
array	必需。规定输入的数组。
value1	必需。规定插入的值。
value2	可选。规定插入的值。
value3	可选。规定插入的值。

提示和注释

注释：所有的数值键名将修改为从零开始重新计数，所有的字符串键名保持不变。

例子 1

```
<?php
$a=array("a"=>"Cat","b"=>"Dog");
array_unshift($a,"Horse");
print_r($a);
?>
```

输出：

```
Array ( [0] => Horse [a] => Cat [b] => Dog )
```

例子 2

返回键值：

```
<?php
$a=array("a"=>"Cat","b"=>"Dog");
print_r(array_unshift($a,"Horse"));
?>
```

输出：

```
3
```

例子 3

数组带有数值键：

```
<?php
$a=array(0=>"Cat",1=>"Dog");
array_unshift($a,"Horse");
print_r($a);
?>
```

输出：

```
Array ( [0] => Horse [1] => Cat [2] => Dog )
```

PHP array_values() 函数

定义和用法

array_values() 函数返回一个包含给定数组中所有键值的数组，但不保留键名。

语法

```
array_values(array)
```

参数	描述
array	必需。规定给定的数组。

提示和注释

提示：被返回的数组将使用数值键，从 0 开始且以 1 递增。

例子

```
<?php
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
print_r(array_values($a));
?>
```

输出：

```
Array ( [0] => Cat [1] => Dog [2] => Horse )
```

PHP array_walk() 函数

定义和用法

array_walk() 函数对数组中的每个元素应用回调函数。如果成功则返回 TRUE，否则返回 FALSE。

典型情况下 *function* 接受两个参数。*array* 参数的值作为第一个，键名作为第二个。如果提供了可选参数 *userdata* ，将被作为第三个参数传递给回调函数。

如果 *function* 函数需要的参数比给出的多，则每次 array*walk()* 调用 _*function* 时都会产生一个 E_WARNING 级的错误。这些警告可以通过在 array_walk() 调用前加上 PHP 的错误操作符 @ 来抑制，或者用 error_reporting()。

语法

```
array_walk(array,function,userdata...)
```

参数	描述
array	必需。规定数组。
function	必需。用户自定义函数的名称。
userdata	可选。用户输入的值，可作为回调函数的参数。

提示和注释

提示：您可以为函数设置一个或多个参数。

注释：如果回调函数需要直接作用于数组中的值，可以将回调函数的第一个参数指定为引用：&$value。（参见例子 3）

注释：将键名和 userdata 传递到 function 中是 PHP 4.0 新增加的。

例子 1

```
<?php
function myfunction($value,$key)
{
echo "The key $key has the value $value<br />";
}
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
array_walk($a,"myfunction");
?>
```

输出：

```
The key a has the value Cat
The key b has the value Dog
The key c has the value Horse
```

例子 2

带有一个参数：

```
<?php
function myfunction($value,$key,$p)
{
echo "$key $p $value<br />";
}
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
array_walk($a,"myfunction","has the value");
?>
```

输出：

```
a has the value Cat
b has the value Dog
c has the value Horse
```

例子 3

改变数组元素的值（请注意 &$value）：

```
<?php
function myfunction(&$value,$key)
{
$value="Bird;
}
$a=array("a"=>"Cat","b"=>"Dog","c"=>"Horse");
array_walk($a,"myfunction");
print_r($a);
?>
```

输出：

```
Array ( [a] => Bird [b] => Bird [c] => Bird )
```

PHP array_walk_recursive() 函数

定义和用法

与 array_walk() 函数 类似，array_walk_recursive() 函数对数组中的每个元素应用回调函数。不一样的是，如果原数组中的元素也是数组，就会递归地调用回调函数，也就是说，会递归到更深层的数组中去。

典型情况下，*function* 接受两个参数。*array* 参数的值作为第一个，键名作为第二个。如果提供了可选参数 *userdata*，将被作为第三个参数传递给回调函数。

如果回调函数需要直接作用于数组中的值，可以将回调函数的第一个参数指定为引用，这样对这些单元的任何改变也将会改变原始数组本身。

语法

```
array_walk_recursive(array,function,userdata)
```

参数	描述
array	必需。规定数组。
function	必需。用户自定义函数的名称。
userdata	可选。用户输入的值，可作为回调函数的参数。

例子

```
<?php
function myfunction($value,$key)
{
echo "The key $key has the value $value<br />";
}
$a1=array("a"=>"Cat","b"=>"Dog");
$a2=array($a1,"1"=>"Bird","2"=>"Horse");
array_walk_recursive($a2,"myfunction");
?>
```

输出：

```
The key a has the value Cat
The key b has the value Dog
The key 1 has the value Bird
The key 2 has the value Horse
```

PHP arsort() 函数

定义和用法

arsort() 函数对数组进行逆向排序并保持索引关系。主要用于对那些单元顺序很重要的结合数组进行排序。

可选的第二个参数包含了附加的排序标识。

如果成功则返回 TRUE，否则返回 FALSE。

语法

```
arsort(array,sorttype)
```

参数	描述
array	必需。输入的数组。
sorttype	可选。规定如何排列数组的值。可能的值： SORT_REGULAR - 默认。以它们原来的类型进行处理（不改变类型）。 SORT_NUMERIC - 把值作为数字来处理 SORT_LOCALE_STRING - 把值作为字符串来处理，基于本地设置*。

*：该值是 PHP 4.4.0 和 5.0.2 新加的。在 PHP 6 之前，使用了系统的区域设置，可以用 setlocale() 来改变。自 PHP 6 起，必须用 i18n_loc_set_default() 函数。

例子

```
<?php
$my_array = array("a" => "Dog", "b" => "Cat", "c" => "Horse");

arsort($my_array);
print_r($my_array);
?>
```

输出：

```
Array
(
[c] => Horse
[a] => Dog
[b] => Cat
)
```

PHP asort() 函数

定义和用法

asort() 函数对数组进行排序并保持索引关系。主要用于对那些单元顺序很重要的结合数组进行排序。

可选的第二个参数包含了附加的排序标识。

如果成功则返回 TRUE，否则返回 FALSE。

语法

```
asort(array,sorttype)
```

参数	描述
array	必需。输入的数组。
sorttype	可选。规定如何排列数组的值。可能的值： `SORT_REGULAR` - 默认。以它们原来的类型进行处理（不改变类型）。 `SORT_NUMERIC` - 把值作为数字来处理 `SORT_STRING` - 把值作为字符串来处理 `SORT_LOCALE_STRING` - 把值作为字符串来处理，基于本地设置*。

*：该值是 PHP 4.4.0 和 5.0.2 新加的。在 PHP 6 之前，使用了系统的区域设置，可以用 setlocale() 来改变。自 PHP 6 起，必须用 i18n_loc_set_default() 函数。

例子

```
<?php
$my_array = array("a" => "Dog", "b" => "Cat", "c" => "Horse");

asort($my_array);
print_r($my_array);
?>
```

输出：

```
Array
(
[b] => Cat
[a] => Dog
[c] => Horse
)
```

PHP compact() 函数

定义和用法

compact() 函数创建一个由参数所带变量组成的数组。如果参数中存在数组，该数组中变量的值也会被获取。

本函数返回的数组是一个关联数组，键名为函数的参数，键值为参数中变量的值。

本函数执行的行为与 extract() 正好相反。

语法

```
compact(var1,var2...)
```

参数	描述
var1	必需。可以是带有变量名的字符串，或者是一个变量数组。
var2	可选。可以是带有变量名的字符串，或者是一个变量数组。允许多个参数。

注释和提示

注释：任何没有变量名与之对应的字符串都被略过。

例子 1

```
<?php
$firstname = "Peter";
$lastname = "Griffin";
$age = "38";

$result = compact("firstname", "lastname", "age");

print_r($result);
?>
```

输出：

```
Array
(
[firstname] => Peter
[lastname] => Griffin
[age] => 38
)
```

例子 2

使用没有对应变量名的字符串，以及一个变量名数组：

```
<?php
$firstname = "Peter";
$lastname = "Griffin";
$age = "38";

$name = array("firstname", "lastname");
$result = compact($name, "location", "age");

print_r($result);
?>
```

输出：

```
Array
(
[firstname] => Peter
[lastname] => Griffin
[age] => 38
)
```

PHP count() 函数

定义和用法

count() 函数计算数组中的单元数目或对象中的属性个数。

对于数组，返回其元素的个数，对于其他值，返回 1。如果参数是变量而变量没有定义，则返回 0。如果 mode 被设置为 COUNT_RECURSIVE（或 1），则会递归底计算多维数组中的数组的元素个数。

语法

```
count(array,mode)
```

参数	描述
array	必需。规定要计数的数组或对象。
mode	可选。规定函数的模式。可能的值： `0` - 默认。不检测多维数组（数组中的数组）。 `1` - 检测多维数组。注释：该参数是 PHP 4.2 中加入的。

提示和注释

注释：当变量未被设置，或是变量包含一个空的数组，该函数会返回 0。可使用 isset() 变量来测试变量是否被设置。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");
$result = count($people);

echo $result;
?>
```

输出：

```
4
```

PHP current() 函数

定义和用法

current() 函数返回数组中的当前元素（单元）。

每个数组中都有一个内部的指针指向它“当前的”元素，初始指向插入到数组中的第一个元素。

current() 函数返回当前被内部指针指向的数组元素的值，并不移动指针。如果内部指针指向超出了单元列表的末端，current() 返回 FALSE。

语法

```
current(array)
```

参数	描述
array	必需。规定要使用的数组。

提示和注释

注释：如果有空的元素，或元素没有值，该函数也返回 FALSE。

提示：该函数不会移动内部指针。要做到这一点，请使用 next() 和 prev() 函数。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");

echo current($people) . "<br />";
?>
```

输出：

```
Peter
```

PHP each() 函数

定义和用法

each() 函数生成一个由数组当前内部指针所指向的元素的键名和键值组成的数组，并把内部指针向前移动。

返回的数组中包括的四个元素：键名为 0，1，key 和 value。单元 0 和 key 包含有数组单元的键名，1 和 value 包含有数据。

如果内部指针越过了数组范围，本函数将返回 FALSE。

语法

```
each(array)
```

参数	描述
array	必需。规定要使用的数组。

例子 1

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");
print_r (each($people));
?>
```

输出：

```
Array ( [1] => Peter [value] => Peter [0] => 0 [key] => 0 )
```

例子 2

each() 经常和 list() 结合使用来遍历数组。本例与上例类似，不过循环输出了整个数组：

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");

reset($people);

while (list($key, $val) = each($people))
  {
  echo "$key => $val<br />";
  }
?>
```

输出：

```
0 => Peter
1 => Joe
2 => Glenn
3 => Cleveland
```

例子解释

因为将一个数组赋值给另一个数组时会重置原来的数组指针，因此在上例中如果我们在循环内部将 $people 赋给了另一个变量的话将会导致无限循环。

PHP extract() 函数

定义和用法

PHP extract() 函数从数组中把变量导入到当前的符号表中。

对于数组中的每个元素，键名用于变量名，键值用于变量值。

第二个参数 type 用于指定当某个变量已经存在，而数组中又有同名元素时，extract() 函数如何对待这样的冲突。

本函数返回成功设置的变量数目。

语法

```
extract(array,extract_rules,prefix)
```

参数	描述
array	必需。规定要使用的输入。
extract_rules	可选。extract() 函数将检查每个键名是否为合法的变量名，同时也检查和符号表中的变量名是否冲突。对非法、数字和冲突的键名的处理将根据此参数决定。可以是以下值之一：可能的值：EXTR_OVERWRITE - 默认。如果有冲突，则覆盖已有的变量。 `EXTR_SKIP` - 如果有冲突，不覆盖已有的变量。（忽略数组中同名的元素） `EXTR_PREFIX_SAME` - 如果有冲突，在变量名前加上前缀 prefix。自 PHP 4.0.5 起，这也包括了对数字索引的处理。 `EXTR_PREFIX_ALL` - 给所有变量名加上前缀 prefix（第三个参数）。 `EXTR_PREFIX_INVALID` - 仅在非法或数字变量名前加上前缀 prefix。本标记是 PHP 4.0.5 新加的。 `EXTR_IF_EXISTS` - 仅在当前符号表中已有同名变量时，覆盖它们的值。其它的都不处理。可以用在已经定义了一组合法的变量，然后要从一个数组例如 $_REQUEST 中提取值覆盖这些变量的场合。本标记是 PHP 4.2.0 新加的。 `EXTR_PREFIX_IF_EXISTS` - 仅在当前符号表中已有同名变量时，建立附加了前缀的变量名，其它的都不处理。本标记是 PHP 4.2.0 新加的。 `EXTR_REFS` - 将变量作为引用提取。这有力地表明了导入的变量仍然引用了 var_array 参数的值。可以单独使用这个标志或者在 extract_type 中用 OR 与其它任何标志结合使用。本标记是 PHP 4.3.0 新加的。
prefix	可选。请注意 prefix 仅在 extract_type 的值是 `EXTR_PREFIX_SAME`，`EXTR_PREFIX_ALL`，`EXTR_PREFIX_INVALID` 或 `EXTR_PREFIX_IF_EXISTS` 时需要。如果附加了前缀后的结果不是合法的变量名，将不会导入到符号表中。前缀和数组键名之间会自动加上一个下划线。

例子 1

```
<?php
$a = 'Original';
$my_array = array("a" => "Cat","b" => "Dog", "c" => "Horse");
extract($my_array);
echo "\$a = $a; \$b = $b; \$c = $c";
?>
```

输出：

```
$a = Cat; $b = Dog; $c = Horse
```

例子 2

使用全部参数：

```
<?php
$a = 'Original';
$my_array = array("a" => "Cat","b" => "Dog", "c" => "Horse");

extract($my_array, EXTR_PREFIX_SAME, 'dup');

echo "\$a = $a; \$b = $b; \$c = $c; \$dup_a = $dup_a;";
?>
```

输出：

```
$a = Original; $b = Dog; $c = Horse; $dup_a = Cat;
```

PHP in_array() 函数

定义和用法

in_array() 函数在数组中搜索给定的值。

语法

```
in_array(value,array,type)
```

参数	描述
value	必需。规定要在数组搜索的值。
array	必需。规定要搜索的数组。
type	可选。如果设置该参数为 true，则检查搜索的数据与数组的值的类型是否相同。

说明

如果给定的值 *value* 存在于数组 *array* 中则返回 true。如果第三个参数设置为 true，函数只有在元素存在于数组中且数据类型与给定值相同时才返回 true。

如果没有在数组中找到参数，函数返回 false。

注释：如果 *value* 参数是字符串，且 *type* 参数设置为 true，则搜索区分大小写。

例子 1

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");

if (in_array("Glenn",$people))
  {
  echo "Match found";
  }
else
  {
  echo "Match not found";
  }
?>
```

输出：

```
Match found
```

例子 2

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland", 23);

if (in_array("23",$people, TRUE))
  {
  echo "Match found<br />";
  }
else
  {
  echo "Match not found<br />";
  }if (in_array("Glenn",$people, TRUE))
  {
  echo "Match found<br />";
  }
else
  {
  echo "Match not found<br />";
  }if (in_array(23,$people, TRUE))
  {
  echo "Match found<br />";
  }
else
  {
  echo "Match not found<br />";
  }
?>
```

输出：

```
Match not found
Match found
Match found
```

PHP key() 函数

定义和用法

key() 函数返回数组内部指针当前指向元素的键名。

若失败，则返回 FALSE。

该函数与 current() 类似，只是返回的结果不同。current() 函数返回的是元素的值，而 key() 函数返回的是元素的键名。

语法

```
key(array)
```

参数	描述
array	必需。规定要使用的数组。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");
echo "The key from the current position is: " . key($people);
?>
```

输出：

```
The key from the current position is: 0
```

PHP krsort() 函数

定义和用法

krsort() 函数将数组按照键逆向排序，为数组值保留原来的键。

可选的第二个参数包含附加的排序标志。

若成功，则返回 TRUE，否则返回 FALSE。

语法

```
krsort(array,sorttype)
```

参数	描述
array	必需。规定要排序的数组。
sorttype	可选。规定如何排列数组的值。可能的值： SORT_REGULAR - 默认。以它们原来的类型进行处理（不改变类型）。 SORT_NUMERIC - 把值作为数字来处理 SORT_STRING - 把值作为字符串来处理 SORT_LOCALE_STRING - 把值作为字符串来处理，基于本地设置*。

*：该值是 PHP 4.4.0 和 5.0.2 新加的。在 PHP 6 之前，使用了系统的区域设置，可以用 setlocale() 来改变。自 PHP 6 起，必须用 i18n_loc_set_default() 函数。

例子

```
<?php
$my_array = array("a" => "Dog", "b" => "Cat", "c" => "Horse");
krsort($my_array);
print_r($my_array);
?>
```

输出：

```
Array
(
[c] => Horse
[b] => Cat
[a] => Dog
)
```

PHP ksort() 函数

定义和用法

ksort() 函数按照键名对数组排序，为数组值保留原来的键。

可选的第二个参数包含附加的排序标志。

若成功，则返回 TRUE，否则返回 FALSE。

语法

```
ksort(array,sorttype)
```

参数	描述
array	必需。规定要排序的数组。
sorttype	可选。规定如何排列数组的值。可能的值： SORT_REGULAR - 默认。以它们原来的类型进行处理（不改变类型）。 SORT_NUMERIC - 把值作为数字来处理 SORT_STRING - 把值作为字符串来处理 SORT_LOCALE_STRING - 把值作为字符串来处理，基于本地设置*。

*：该值是 PHP 4.4.0 和 5.0.2 新加的。在 PHP 6 之前，使用了系统的区域设置，可以用 setlocale() 来改变。自 PHP 6 起，必须用 i18n_loc_set_default() 函数。

例子

```
<?php
$my_array = array("a" => "Dog", "b" => "Cat", "c" => "Horse");
ksort($my_array);
print_r($my_array);
?>
```

输出：

```
Array
(
[a] => Dog
[b] => Cat
[c] => Horse
)
```

PHP list() 函数

定义和用法

list() 函数用数组中的元素为一组变量赋值。

注意，与 array() 类似，list() 实际上是一种语言结构，不是函数。

语法

```
list(var1,var2...)
```

参数	描述
var1	必需。第一个需要赋值的变量。
var2	可选。可以有多个变量。

提示和注释

注释：该函数只用于数字索引的数组，且假定数字索引从 0 开始。

例子 1

```
<?php
$my_array = array("Dog","Cat","Horse");

list($a, $b, $c) = $my_array;
echo "I have several animals, a $a, a $b and a $c.";
?>
```

输出：

```
I have several animals, a Dog, a Cat and a Horse.
```

例子 2

```
<?php
$my_array = array("Dog","Cat","Horse");

list($a, , $c) = $my_array;
echo "Here I only use the $a and $c variables.";
?>
```

输出：

```
Here I only use the Dog and Horse variables.
```

PHP natcasesort() 函数

定义和用法

natcasesort() 函数用不区分大小写的自然顺序算法对给定数组中的元素排序。

natcasesort() 函数实现了“自然排序”，即数字从 1 到 9 的排序方法，字母从 a 到 z 的排序方法，短者优先，该函数不区分大小写。数组的索引与单元值保持关联。

如果成功，则该函数返回 TRUE，否则返回 FALSE。

语法

```
natcasesort(array)
```

参数	描述
array	必需。规定要进行排序的数组。

提示和注释

提示：natcasesort() 是 natsort() 函数 的不区分大小写字母的版本。

例子

```
<?php
$temp_files = array("temp15.txt","Temp10.txt",
"temp1.txt","Temp22.txt","temp2.txt");

natsort($temp_files);
echo "Natural order: ";
print_r($temp_files);
echo "<br />";

natcasesort($temp_files);
echo "Natural order case insensitve: ";
print_r($temp_files);
?>
```

输出：

```
Natural order:
Array
(
[0] => Temp10.txt
[1] => Temp22.txt
[2] => temp1.txt
[4] => temp2.txt
[3] => temp15.txt
)

Natural order case insensitve:
Array
(
[2] => temp1.txt
[4] => temp2.txt
[0] => Temp10.txt
[3] => temp15.txt
[1] => Temp22.txt
)
```

PHP natsort() 函数

定义和用法

natsort() 函数用自然顺序算法对给定数组中的元素排序。

natsort() 函数实现了“自然排序”，即数字从 1 到 9 的排序方法，字母从 a 到 z 的排序方法，短者优先。数组的索引与单元值保持关联。

如果成功，则该函数返回 TRUE，否则返回 FALSE。

语法

```
natsort(array)
```

参数	描述
array	必需。规定要进行排序的数组。

例子

本函数所用的自然排序算法，与通常的计算机字符串排序算法（用于 sort()）的区别，见下面示例：

```
<?php
$temp_files = array("temp15.txt","temp10.txt",
"temp1.txt","temp22.txt","temp2.txt");

sort($temp_files);
echo "Standard sorting: ";
print_r($temp_files);
echo "<br />";

natsort($temp_files);
echo "Natural order: ";
print_r($temp_files);
?>
```

输出：

```
Standard sorting: Array
(
[0] => temp1.txt
[1] => temp10.txt
[2] => temp15.txt
[3] => temp2.txt
[4] => temp22.txt
)

Natural order: Array
(
[0] => temp1.txt
[3] => temp2.txt
[1] => temp10.txt
[2] => temp15.txt
[4] => temp22.txt
)
```

PHP next() 函数

定义和用法

next() 函数把指向当前元素的指针移动到下一个元素的位置，并返回当前元素的值。

如果内部指针已经超过数组的最后一个元素，函数返回 false。

语法

```
next(array)
```

参数	描述
array	必需。规定要使用的数组。

说明

next() 和 current() 的行为类似，只有一点区别，在返回值之前将内部指针向前移动一位。这意味着它返回的是下一个数组单元的值并将数组指针向前移动了一位。如果移动指针的结果超出了数组单元的末端，则 next() 返回 FALSE。

注意：如果数组包含空的单元，或者单元的值是 0 则该函数碰到这些单元也返回 FALSE。要正确遍历可能含有空单元或者单元值为 0 的数组，请参见 each() 函数。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");

echo current($people) . "<br />";
echo next($people);
?>
```

输出：

```
Peter
Joe
```

PHP pos() 函数

定义和用法

pos() 函数是 current() 函数 的别名。它可返回数组中当前元素的值。

语法

```
pos(array)
```

参数	描述
array	必需。规定要使用的数组。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");
echo pos($people) . "<br />";
?>
```

输出：

```
Peter
```

PHP prev() 函数

定义和用法

prev() 函数把指向当前元素的指针移动到上一个元素的位置，并返回当前元素的值。

如果内部指针已经超过数组的第一个元素之前，函数返回 false。

语法

```
prev(_array_)
```

参数	描述
array	必需。规定要使用的数组。

说明

prev() 和 next() 的行为类似，不过它将内部指针倒回一位而不是前移一位。

注意：如果数组包含空的单元，或者单元的值是 0 则该函数碰到这些单元也返回 FALSE。要正确遍历可能含有空单元或者单元值为 0 的数组，请参见 each() 函数。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");

echo current($people) . "<br />";
echo next($people) . "<br />";
echo prev($people);
?>
```

输出：

```
Peter
Joe
Peter
```

PHP range() 函数

定义和用法

range() 函数创建并返回一个包含指定范围的元素的数组。

语法

```
range(first,second,step)
```

参数	描述
first	必需。规定数组元素的最小值。
second	必需。规定数组元素的最大值。
step	可选。规定元素之间的步进制。默认是 1。注释：该参数是 PHP 5 中加入的。

说明

该函数创建一个数组，包含从 first 到 second （包含 first 和 second）之间的整数或字符。如果 second 比 first 小，则返回反序的数组。

例子 1

```
<?php
$number = range(0,5);
print_r ($number);
?>
```

输出：

```
Array
(
[0] => 0
[1] => 1
[2] => 2
[3] => 3
[4] => 4
[5] => 5
)
```

例子 2

```
<?php
$number = range(0,50,10);
print_r ($number);
?>
```

输出：

```
Array
(
[0] => 0
[1] => 10
[2] => 20
[3] => 30
[4] => 40
[5] => 50
)
```

例子 3

```
<?php
$letter = range("a","d");
print_r ($letter);
?>
```

输出：

```
Array
(
[0] => a
[1] => b
[2] => c
[3] => d
)
```

PHP reset() 函数

定义和用法

reset() 函数把数组的内部指针指向第一个元素，并返回这个元素的值。

若失败，则返回 FALSE。

语法

```
reset(array)
```

参数	描述
array	必需。规定要使用的数组。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");

echo current($people) . "<br />";
echo next($people) . "<br />";
echo reset($people);
?>
```

输出：

```
Peter
Joe
Peter
```

PHP rsort() 函数

定义和用法

rsort() 函数对数组的元素按照键值进行逆向排序。与 arsort() 的功能基本相同。

注释：该函数为 array 中的单元赋予新的键名。这将删除原有的键名而不仅是重新排序。

如果成功则返回 TRUE，否则返回 FALSE。

可选的第二个参数包含另外的排序标志。

语法

```
rsort(array,sorttype)
```

参数	描述
array	必需。输入的数组。
sorttype	可选。规定如何排列数组的值。可能的值： SORT_REGULAR - 默认。以它们原来的类型进行处理（不改变类型）。 SORT_NUMERIC - 把值作为数字来处理 SORT_STRING - 把值作为字符串来处理 SORT_LOCALE_STRING - 把值作为字符串来处理，基于本地设置*。

*：该值是 PHP 4.4.0 和 5.0.2 新加的。在 PHP 6 之前，使用了系统的区域设置，可以用 setlocale() 来改变。自 PHP 6 起，必须用 i18n_loc_set_default() 函数。

例子

```
<?php
$my_array = array("a" => "Dog", "b" => "Cat", "c" => "Horse");
rsort($my_array);
print_r($my_array);
?>
```

输出：

```
Array
(
[0] => Horse
[1] => Dog
[2] => Cat
)
```

PHP shuffle() 函数

定义和用法

shuffle() 函数把数组中的元素按随机顺序重新排列。

若成功，则返回 TRUE，否则返回 FALSE。

注释：本函数为数组中的单元赋予新的键名。这将删除原有的键名而不仅是重新排序。

注释：自 PHP 4.2.0 起，不再需要用 srand() 或 mt_srand() 函数给随机数发生器播种，现已被自动完成。

语法

```
shuffle(array)
```

参数	描述
array	必需。规定要使用的数组。

例子

```
<?php
$my_array = array("a" => "Dog", "b" => "Cat", "c" => "Horse");
shuffle($my_array);
print_r($my_array);
?>
```

输出：

```
Array ( [0] => Cat [1] => Horse [2] => Dog )
```

PHP sizeof() 函数

定义和用法

sizeof() 函数计算数组中的单元数目或对象中的属性个数。

该函数是 count() 的别名。

语法

```
sizeof(array,mode)
```

参数	描述
array	必需。规定要计数的数组或对象。
mode	可选。规定函数的模式。可能的值： `0` - 默认。不检测多维数组（数组中的数组）。 `1` - 检测多维数组。注释：该参数是 PHP 4.2 中加入的。

提示和注释

注释：当变量未被设置，或是变量包含一个空的数组，该函数会返回 0。可使用 isset() 变量来测试变量是否被设置。

例子

```
<?php
$people = array("Peter", "Joe", "Glenn", "Cleveland");
$result = sizeof($people);

echo $result;
?>
```

输出：

```
4
```

PHP sort() 函数

定义和用法

sort() 函数按升序对给定数组的值排序。

注释：本函数为数组中的单元赋予新的键名。原有的键名将被删除。

如果成功则返回 TRUE，否则返回 FALSE。

语法

```
sort(_array_,_sorttype_)
```

参数	描述
array	必需。输入的数组。
sorttype	可选。规定如何排列数组的值。可能的值： `SORT_REGULAR` - 默认。以它们原来的类型进行处理（不改变类型）。 `SORT_NUMERIC` - 把值作为数字来处理 `SORT_STRING` - 把值作为字符串来处理 `SORT_LOCALE_STRING` - 把值作为字符串来处理，基于本地设置*。

*：该值是 PHP 4.4.0 和 5.0.2 新加的。在 PHP 6 之前，使用了系统的区域设置，可以用 setlocale() 来改变。自 PHP 6 起，必须用 i18n_loc_set_default() 函数。

例子

```
<?php
$my_array = array("a" => "Dog", "b" => "Cat", "c" => "Horse");

sort($my_array);
print_r($my_array);
?>
```

输出：

```
Array
(
[0] => Cat
[1] => Dog
[2] => Horse
)
```

PHP uasort() 函数

定义和用法

uasort() 函数使用用户自定义的比较函数对数组排序，并保持索引关联（不为元素分配新的键）。

如果成功则返回 TRUE，否则返回 FALSE。

该函数主要用于对那些单元顺序很重要的结合数组进行排序。

语法

```
uasort(array,sorttype)
```

参数	描述
array	必需。规定要排序的数组。
function	

必需。用户自定义的函数。

函数必须设计为返回 -1, 0, 或 1，并应该接受两个供比较的参数，同时以类似下面这样的方式来工作： 如果 a = b, 返回 0； 如果 a < b, 返回 1； 如果 a > b, 返回 -1。 |

例子

```
<?php
function my_sort($a, $b)
  {
  if ($a == $b) return 0;
  return ($a > $b) ? -1 : 1;
  }

$people = array("Swanson" => "Joe",
"Griffin" => "Peter", "Quagmire" => "Glenn",
"swanson" => "joe", "griffin" => "peter",
"quagmire" => "glenn");

uasort($people, "my_sort");

print_r ($people);
?>
```

输出：

```
Array
(
[griffin] => peter
[swanson] => joe
[quagmire] => glenn
[Griffin] => Peter
[Swanson] => Joe
[Quagmire] => Glenn
)
```

PHP uksort() 函数

定义和用法

uksort() 函数使用用户自定义的比较函数按照键名对数组排序，并保持索引关系。

如果成功则返回 TRUE，否则返回 FALSE。

如果要排序的数组需要用一种不寻常的标准进行排序，那么应该使用此函数。

自定义函数应接受两个参数，该参数将被数组中的一对键名填充。比较函数在第一个参数小于，等于，或大于第二个参数时必须分别返回一个小于零，等于零，或大于零的整数。

语法

```
uksort(array,sorttype)
```

参数	描述
array	必需。规定要排序的数组。
function	必需。用户自定义的函数。函数必须设计为返回 -1, 0, 或 1，并应该接受两个供比较的参数，同时以类似下面这样的方式来工作： 如果 a = b, 返回 0； 如果 a > b, 返回 1； 如果 a < b, 返回 -1。

例子

```
<?php
function my_sort($a, $b)
  {
  if ($a == $b) return 0;
  return ($a > $b) ? -1 : 1;
  }

$people = array("Swanson" => "Joe",
"Griffin" => "Peter", "Quagmire" => "Glenn",
"swanson" => "joe", "griffin" => "peter",
"quagmire" => "glenn");

uksort($people, "my_sort");

print_r ($people);
?>
```

输出：

```
Array
(
[swanson] => joe
[quagmire] => glenn
[griffin] => peter
[Swanson] => Joe
[Quagmire] => Glenn
[Griffin] => Peter
)
```

PHP usort() 函数

定义和用法

usort() 函数使用用户自定义的函数对数组排序。

注释：如果两个元素比较结果相同，则它们在排序后的数组中的顺序未经定义。到 PHP 4.0.6 之前，用户自定义函数将保留这些元素的原有顺序。但是由于在 4.1.0 中引进了新的排序算法，结果将不是这样了，因为对此没有一个有效的解决方案。

注释：本函数为 *array* 中的元素赋予新的键名。这会删除原有的键名。

语法

```
usort(array,sorttype)
```

参数	描述
array	必需。规定要排序的数组。
function	必需。用户自定义的函数。函数必须设计为返回 -1, 0, 或 1，并应该接受两个供比较的参数，同时以类似下面这样的方式来工作： 如果 a = b, 返回 0； 如果 a > b, 返回 1； 如果 a < b, 返回 -1。

例子

```
<?php
function my_sort($a, $b)
  {
  if ($a == $b) return 0;
  return ($a > $b) ? -1 : 1;
  }

$arr = array("Peter", "glenn","Cleveland","peter","cleveland", "Glenn");

usort($arr, "my_sort");

print_r ($arr);
?>
```

输出：

```
Array
(
[0] => peter
[1] => glenn
[2] => cleveland
[3] => Peter
[4] => Glenn
[5] => Cleveland
)
```

PHP Calendar 函数

PHP Calendar 简介

当使用不同的历法格式时，calendar 函数很有用。它所基于的标准是儒略日计数 (Julian day count)。

编者注：Julian day count 是从 January 1, 4713 B.C. 开始计算的，中文译为儒略日计数或恺撒日计数。

请注意，Julian day count（儒略日计数） 与 Julian calendar（儒略历） 不是一回事。

注释：如需在日历格式之间转换，必须首先转换为 Julian day count，然后再转换为日历格式。

安装

PHP 的 windows 版本已内建了对日历扩展的支持。因此，Calendar 函数会自动工作。

不过，如果您运行的是 PHP 的 Linux 版本，就不得不通过 --enable-calendar 编译 PHP，这样日历函数才能工作。

PHP Calendar 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
cal_days_in_month()	针对指定的年份和日历，返回一个月中的天数。	4
cal_from_jd()	把儒略日计数转换为指定日历的日期。	4
cal_info()	返回有关给定日历的信息。	4
cal_to_jd()	把日期转换为儒略日计数。	4
easter_date()	返回指定年份的复活节午夜的 Unix 时间戳。	3
easter_days()	返回指定年份的复活节与 3 月 21 日之间的天数。	3
FrenchToJD()	将法国共和历法转换成为儒略日计数。	3
GregorianToJD()	将格利高里历法转换成为儒略日计数。	3
JDDayOfWeek()	返回日期在周几。	3
JDMonthName()	返回月的名称。	3
JDToFrench()	把儒略日计数转换为法国共和国历法。	3
JDToGregorian()	把儒略日计数转换为格利高里历法。	3
jdtojewish()	把儒略日计数转换为犹太历法。	3
JDToJulian()	把儒略日计数转换为儒略历。	3
jdtounix()	把儒略日计数转换为 Unix 时间戳。	4
JewishToJD()	把犹太历法转换为儒略日计数。	3
JulianToJD()	把儒略历转换为儒略日计数。	3
unixtojd()	把 Unix 时间戳转换为儒略日计数。	4

PHP Calendar 常量

PHP：指示支持该常量的最早的 PHP 版本。

常量	描述	PHP
CAL_GREGORIAN	Gregorian calendar	3
CAL_JULIAN	Julian calendar	3
CAL_JEWISH	Jewish calendar	3
CAL_FRENCH	French Republican calendar	3
CAL_NUM_CALS	3	
CAL_DOW_DAYNO	3	
CAL_DOW_SHORT	3	
CAL_DOW_LONG	3	
CAL_MONTH_GREGORIAN_SHORT	3	
CAL_MONTH_GREGORIAN_LONG	3	
CAL_MONTH_JULIAN_SHORT	3	
CAL_MONTH_JULIAN_LONG	3	
CAL_MONTH_JEWISH	3	
CAL_MONTH_FRENCH	3	
CAL_EASTER_DEFAULT	4	
CAL_EASTER_DEFAULT	4	
CAL_EASTER_ROMAN	4	
CAL_EASTER_ALWAYS_GREGORIAN	4	
CAL_EASTER_ALWAYS_JULIAN	4	
CAL_JEWISH_ADD_ALAFIM_GERESH	5	
CAL_JEWISH_ADD_ALAFIM	5	
CAL_JEWISH_ADD_GERESHAYIM	5	

PHP cal_days_in_month() 函数

定义和用法

cal_days_in_month() 函数针对指定的年份和日历，返回一个月中的天数。

语法

```
cal_days_in_month(calendar,month,year)
```

参数	描述
calendar	必需。规定要使用的历法。
month	必须。规定月。
year	必须。规定年。

例子

```
<?php
$d=cal_days_in_month(CAL_GREGORIAN,10,2005);
echo("There was $d days in October 2005");
?>
```

输出：

```
There was 31 days in October 2005
```

PHP cal_from_jd() 函数

定义和用法

cal_from_jd() 函数把儒略日计数转换为指定历法的日期。

语法

```
cal_from_jd(jd,calendar)
```

参数	描述
jd	必需。一个数字（儒略日计数）。
calendar	必需。规定要使用的历法。可以使用下面这些常量： `CAL_GREGORIAN` `CAL_JULIAN` `CAL_JEWISH` `CAL_FRENCH`

例子

```
<?php
$d=unixtojd(mktime(0,0,0,1,18,2006));
print_r(cal_from_jd($d,CAL_GREGORIAN));
?>
```

输出：

```
Array
(
[date] => 1/18/2006
[month] => 1
[day] => 18
[year] => 2006
[dow] => 3
[abbrevdayname] => Wed
[dayname] => Wednesday
[abbrevmonth] => Jan
[monthname] => January
)
```

PHP cal_info() 函数

定义和用法

cal_info() 函数返回一个数组，其中包含了关于给定历法的信息。

所返回的数组包含这些元素：calname, calsymbol, month, abbrevmonth 以及 maxdaysinmonth。

语法

```
cal_from_jd(jd,calendar)
```

参数	描述
calendar	必需。规定要使用的历法。可以使用下面这些常量： `CAL_GREGORIAN` `CAL_JULIAN` `CAL_JEWISH` `CAL_FRENCH`

提示和注释

提示：在 PHP 5 中，如果没有指定 calendar 参数，则返回所有被支持的历法的信息。

例子

```
<?php
$calinfo=cal_info(0);
print_r($calinfo);
?>
```

输出：

```
Array
(
[months] => Array
(
[1] => January
[2] => February
[3] => March
[4] => April
[5] => May
[6] => June
[7] => July
[8] => August
[9] => September
[10] => October
[11] => November
[12] => December
)

[abbrevmonths] => Array
(
[1] => Jan
[2] => Feb
[3] => Mar
[4] => Apr
[5] => May
[6] => Jun
[7] => Jul
[8] => Aug
[9] => Sep
[10] => Oct
[11] => Nov
[12] => Dec
)

[maxdaysinmonth] => 31
[calname] => Gregorian
[calsymbol] => CAL_GREGORIAN
)
```

PHP cal_to_jd() 函数

定义和用法

cal_to_jd() 函数把指定的日期转换为儒略日计数。

语法

```
cal_to_jd(calendar,month,day,year)
```

参数	描述
calendar	必需。规定要使用的历法。可以使用下面这些常量：`CAL_GREGORIAN` `CAL_JULIAN` `CAL_JEWISH` `CAL_FRENCH`
month	必需。规定月。
day	必需。规定日。
year	必需。规定年。

例子

```
<?php
$d=cal_to_jd(CAL_GREGORIAN,10,03,2005);
echo($d);
?>
```

输出：

```
2453647
```

PHP easter_date() 函数

定义和用法

easter_date() 函数返回指定年份的复活节午夜的 Unix 时间戳。

输入一个年份，则以 UNIX 时间戳记的格式返回该年的复活节日期，若没有输入年份，则计算当年的日期。

语法

```
easter_date(year)
```

参数	描述
year	可选。定义用于计算复活节日期的年份。若省略，使用当年。

提示和注释

注释：如果年份在 Unix 时间戳的范围之外（1970 之前或 2037 之后），该函数会生成一个警告。可使用 easter_days() 代替 easter_date() 来计算年份在范围之外的复活节日期。

例子

```
<?php
echo(easter_date() . "<br />");
echo(date("M-d-Y",easter_date()) . "<br />");
echo(date("M-d-Y",easter_date(2000)) . "<br />");
echo(date("M-d-Y",easter_date(2001)) . "<br />");
echo(date("M-d-Y",easter_date(2002)));
?>
```

输出：

```
1145138400
Apr-16-2006
Apr-23-2000
Apr-15-2001
Mar-31-2002
```

PHP easter_days() 函数

定义和用法

easter_days() 函数返回指定年份的复活节与 3 月 21 日之间的天数。

输入一个年份，则计算该年复活节与三月二十一日之间的日期数，若没有输入年份，则以当年计算。这个函数可以用来替代 easter_date() 在 1970-2037 年范围外无法计算的问题。

语法

```
easter_date(year)
```

参数	描述
year	可选。定义用于计算复活节日期的年份。若省略，使用当年。
method	可选。允许你计算机与其它历法的复活节日期。例如，如果设置为 CAL_EASTER_ROMAN，则使用 1582 - 1752 年期间的格利高里历法。

例子

```
<?php
echo(easter_days() . "<br />");
echo(easter_days(1990) . "<br />");
echo(easter_days(1342) . "<br />");
echo(easter_days(2050);
?>
```

输出：

```
26
25
10
20
```

PHP FrenchToJD() 函数

定义和用法

FrenchToJD() 函数将法国共和历法转换成为儒略日计数。

语法

```
frenchtojd(month,day,year)
```

参数	描述
month	必需。规定月
day	必需。规定日
year	可选。必须在 1 到 14 的范围内。

提示和注释

法国共和历法是法国革命期间提出的一种历法，从 1793 年晚期开始，法国政府使用了大约 12 年。该函数只转换 1 到 14 年内的日期 （格利高里历 1792 年 9 月 22 日 - 1806 年 9 月 22 日）。

例子

```
<?php
$d=frenchtojd(3,3,14);
echo($d);
?>
```

输出：

```
2380650
```

PHP GregorianToJD() 函数

定义和用法

GregorianToJD() 函数将格利高里历法转换成为儒略日计数。

语法

```
gregoriantojd(month,day,year)
```

参数	描述
month	必需。规定月
day	必需。规定日
year	可选。合法的范围是 4714 B.C. 到 9999 A.D。

提示和注释

尽管该函数可处理 4714 B.C. 之前的日期，您还是要注意格利高里历法在 1582 年才建立，一些国家甚至更晚才接受它（大不列颠在 1752 年，苏联在 1918 年，希腊在 1923 年）。大部分欧洲国家使用罗马儒略历（公历）先于格利高里历法。

例子

```
<?php
$jd = gregoriantojd(10,3,1975);
echo($jd . "<br />");

$gregorian = jdtogregorian($jd);
echo($gregorian);
?>
```

输出：

```
2442689
10/3/1975
```

PHP JDDayOfWeek() 函数

定义和用法

JDDayOfWeek() 函数返回日期在周几。

语法

```
jddayofweek(jd,mode)
```

参数	描述
jd	必需。数字（儒略日计数）。
mode	可选。定义返回的内容（数字还是字符串）。模式值： 0 - 默认。以整数返回周的天。（0 为周日, 1 为周一... 余类推） 1 - 返回包含周的天的字符串。（英文-格里高里历） 2 - 返回包含周的天的简写的字符串。（英文-格里高里历）

例子

```
<?php
$jd=cal_to_jd(CAL_GREGORIAN,date("m"),date("d"),date("Y"));
echo(jddayofweek($jd,1));
?>
```

输出：

```
Thursday
```

PHP JDMonthName() 函数

定义和用法

JDMonthName() 函数返回指定历法的月份字符串。

语法

```
jdmonthname(jd,mode)
```

参数	描述
jd	必需。数字（儒略日计数）。
mode	可选。定义把儒略日计数转换为哪种历法，以及返回哪种月份名称。模式值： `0` - 格里高里历 (缩写) (Jan, Feb, Mar, ...) `1` - 格里高里历 (January, February, March, ...) `2` - 凯撒历 (缩写) (Jan, Feb, Mar, ...) `3` - 凯撒历 (January, February, March, ...) `4` - 犹太历 (Tishri, Heshvan, Kislev, ...) `5` - 法国共和历 (Vendemiaire, Brumaire, Frimaire, ...)

例子

```
<?php
$jd=cal_to_jd(CAL_GREGORIAN,date("m"),date("d"),date("Y"));
echo(jdmonthname($jd,1));
?>
```

输出：

```
January
```

PHP JDToFrench() 函数

定义和用法

JDToFrench() 函数把儒略日计数转换为法国共和国历法。

语法

```
jdtofrench(jd)
```

参数	描述
jd	必需。数字（儒略日计数）。

提示和注释：

注释：该函数以 "month/day/year" 的格式返回一个字符串。

提示：法国共和历法是法国革命期间提出的一种历法，从 1793 年晚期开始，法国政府使用了大约 12 年。

例子

```
<?php
$d=jdtofrench(2380650);
echo($d);
?>
```

输出：

```
3/3/14
```

PHP JDToGregorian() 函数

定义和用法

JDToGregorian() 函数把儒略日计数转换为格利高里历法。

语法

```
jdtogregorian(jd)
```

参数	描述
jd	必需。数字（儒略日计数）。

提示和注释：

注释：该函数以 "month/day/year" 的格式返回一个字符串。

例子

```
<?php
$jd = gregoriantojd(10,3,1975);
echo($jd . "<br />");

$gregorian = jdtogregorian($jd);
echo($gregorian);
?>
```

输出：

```
2442689
10/3/1975
```

PHP JDToJewish() 函数

定义和用法

JDToJewish() 函数把儒略日计数转换为犹太历法

语法

```
JDToJewish(jd)
```

参数	描述
jd	必需。数字（儒略日计数）。
hebrew	可选。True 指示希伯来语输出格式。
fl	可选。定义希伯来语输出格式，可用的格式有：`CAL_JEWISH_ADD_ALAFIM_GERESH` `CAL_JEWISH_ADD_ALAFIM` `CAL_JEWISH_ADD_GERESHAYIM`

例子

```
<?php
echo(jdtojewish(gregoriantojd(10,8,2002)));
?>
```

输出：

```
2/2/5763
```

PHP JDToJulian() 函数

定义和用法

JDToJulian() 函数把儒略日计数转换为儒略历。

语法

```
JDToJulian(jd)
```

参数	描述
jd	必需。数字（儒略日计数）。

提示和注释：

注释：该函数以 "month/day/year" 的格式返回一个字符串。

例子

```
<?php
$jd = juliantojd(10,3,1975);
echo($jd . "<br />");

$julian = jdtojulian($jd);
echo($julian);
?>
```

输出：

```
2442702
10/3/1975
```

PHP JDToUnix() 函数

定义和用法

JDToUnix() 函数把儒略日计数转换为 Unix 时间戳。

语法

```
JDToUnix(jd)
```

参数	描述
jd	必需。数字（儒略日计数）。

提示和注释：

注释：如果参数 jd 不在 Unix 新纪元之中（意味着格利高里年必须介于 1970 和 2037 之间，或者 jd >= 2440588 且 jd <= 2465342），则该函数将返回 false。所返回的时间是本地时间。

例子

```
<?php
$jd = gregoriantojd(10,3,1970);
$unix = jdtounix($jd);
echo($unix);
?>
```

输出：

```
23760000
```

PHP JewishToJD() 函数

定义和用法

JewishToJD() 函数把犹太历法转换为儒略日计数。

语法

```
JDToUnix(jd)
```

参数	描述
month	必需。规定月。
day	必需。规定日。
year	必需。规定年。

提示和注释：

注释：有效的范围为犹太历法公元前 3761 年起。犹太历法存在了数千年，但早期并没有公式化的开始月份计算法。每年的第一个月为首次观测到的新月。

例子

```
<?php
echo(jewishtojd(2,2,5763));
?>
```

输出：

```
2452556
```

PHP JulianToJD() 函数

定义和用法

JulianToJD() 函数把儒略历转换为儒略日计数。

语法

```
JDToUnix(jd)
```

参数	描述
month	必需。规定月。
day	必需。规定日。
year	必需。规定年。合法的范围是 4713 B.C. 到 9999 A.D。

提示和注释：

注释：有效的范围为凯撒历法公元前 4713 年至公元 9999 年。该函数能计算到公元前 4713 年，但这是不太有意义的。凯撒历法是在公元前 46 年建立的，但一些细节等到公元 8 年才稳定下来。

例子

```
<?php
$jd = juliantojd(10,3,1975);
echo($jd . "<br />");

$julian = jdtojulian($jd);
echo($julian);
?>
```

输出：

```
2442702
10/3/1975
```

PHP UnixToJD() 函数

定义和用法

UnixToJD() 函数把 Unix 时间戳转换为儒略日计数。

语法

```
unixtojd(timestamp)
```

参数	描述
timestamp	可选。

提示和注释：

注释：Unix 时间戳指示的是从格利高里历（不是罗马儒略历）的 1970 年 1 月 1 日至今的秒数。

例子

```
<?php
echo(unixtojd());
?>
```

输出：

```
2453755
```

PHP cURL 函数

概述

PHP支持的由Daniel Stenberg创建的libcurl库允许你与各种的服务器使用各种类型的协议进行连接和通讯。

libcurl目前支持http、https、ftp、gopher、telnet、dict、file和ldap协议。libcurl同时也支持HTTPS认证、HTTP POST、HTTP PUT、 FTP 上传(这个也能通过PHP的FTP扩展完成)、HTTP 基于表单的上传、代理、cookies和用户名+密码的认证。

PHP中使用cURL实现Get和Post请求的方法

这些函数在PHP 4.0.2中被引入。

需求

为了使用PHP的cURL函数，你需要安装 ? libcurl包。

PHP需要使用libcurl 7.0.2-beta 或者更高版本。在PHP 4.2.3 里使用cURL，你需要安装7.9.0或更高版本的libcurl。从PHP 4.3.0开始你需要安装7.9.0或更高版本的libcurl。从PHP 5.0.0开始你需要安装7.10.5或更高版本的libcurl。

安装

要使用PHP的cURL支持你必须在编译PHP时加上--with-curl[=DIR] 选项，DIR为包含lib和include的目录路径。在include目录中必须有一个名为curl，包含了easy.h和curl.h的文件夹。lib文件夹里应该有一个名为libcurl.a的文件。对于PHP 4.3.0你可以配置--with-curlwrappers 使cURL使用URL流。

注意: Win32用户注意 要在Windows环境下使用这个模块，libeay32.dll和ssleay32.dll必须放到PATH环境变量包含的目录下。 不用cURL网站上的libcurl.dll。

资源类型

这个扩展定义了2中资源：cURL句柄和cURL批处理句柄。

PHP cURL 函数

以下包含了PHP cURL函数列表：

函数	描述
curl_close()	关闭一个cURL会话。
curl_copy_handle()	复制一个cURL句柄和它的所有选项。
curl_errno()	返回最后一次的错误号。
curl_error()	返回一个保护当前会话最近一次错误的字符串。
curl_escape()	返回转义字符串，对给定的字符串进行URL编码。
curl_exec()	执行一个cURL会话。
curl_file_create()	创建一个 CURLFile 对象。
curl_getinfo()	获取一个cURL连接资源句柄的信息。
curl_init()	初始化一个cURL会话。
curl_multi_add_handle()	向curl批处理会话中添加单独的curl句柄。
curl_multi_close()	关闭一组cURL句柄。
curl_multi_exec()	运行当前 cURL 句柄的子连接。
curl_multi_getcontent()	如果设置了CURLOPT_RETURNTRANSFER，则返回获取的输出的文本流。
curl_multi_info_read()	获取当前解析的cURL的相关传输信息。
curl_multi_init()	返回一个新cURL批处理句柄。
curl_multi_remove_handle()	移除curl批处理句柄资源中的某个句柄资源。
curl_multi_select()	等待所有cURL批处理中的活动连接。
curl_multi_setopt()	设置一个批处理cURL传输选项。
curl_multi_strerror()	返回描述错误码的字符串文本。
curl_pause()	暂停及恢复连接。
curl_reset()	重置libcurl的会话句柄的所有选项。
curl_setopt_array()	为cURL传输会话批量设置选项。
curl_setopt()	设置一个cURL传输选项。
curl_share_close()	关闭cURL共享句柄。
curl_share_init()	初始化cURL共享句柄。
curl_share_setopt()	设置一个共享句柄的cURL传输选项。
curl_strerror()	返回错误代码的字符串描述。
curl_unescape()	解码URL编码后的字符串。
curl_version()	获取cURL版本信息。

PHP curl_close函数

(PHP 4 >= 4.0.2, PHP 5)

curl_close — 关闭一个cURL会话

说明

```
void curl_close ( resource $ch )
```

关闭一个cURL会话并且释放所有资源。cURL句柄ch 也会被释放。

参数

ch

由 curl_init() 返回的 cURL 句柄。

返回值

没有返回值。

实例

初始化一个cURL会话来获取一个网页

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.w3cschool.cc/");
curl_setopt($ch, CURLOPT_HEADER, 0);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close($ch);
?>
```

参见

- curl_init() - 初始化一个cURL会话
- curl_multi_close() - 关闭一组cURL句柄

PHP curl_copy_handle函数

(PHP 5)

curl_copy_handle — 复制一个cURL句柄和它的所有选项

说明

```
resource curl_copy_handle ( resource $ch )
```

复制一个cURL句柄并保持相同的选项。

参数

ch

由 curl_init() 返回的 cURL 句柄。

返回值

返回一个新的cURL句柄。

实例

复制一个cURL句柄

```
<?php
// 创建一个新的cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, 'http://www.example.com/');
curl_setopt($ch, CURLOPT_HEADER, 0);

// 复制句柄
$ch2 = curl_copy_handle($ch);

// 抓取URL (http://www.example.com/) 并把它传递给浏览器
curl_exec($ch2);

// 关闭cURL资源，并且释放系统资源
curl_close($ch2);
curl_close($ch);
?>
```

PHP curl_errno函数

(PHP 4 >= 4.0.3, PHP 5)

curl_errno — 返回最后一次的错误号

说明

```
int curl_errno ( resource $ch )
```

返回最后一次cURL操作的错误号。

参数

ch

由 curl_init() 返回的 cURL 句柄。

返回值

返回错误号或 0 (零) 如果没有错误发生。

实例

```
<?php
// 创建一个指向一个不存在的位置的cURL句柄
$ch = curl_init('http://404.php.net/');

// 执行
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_exec($ch);

// 检查是否有错误发生
if(curl_errno($ch))
{
    echo 'Curl error: ' . curl_error($ch);
}

// 关闭句柄
curl_close($ch);
?>
```

参见

- curl_error() - 返回一个保护当前会话最近一次错误的字符串
- ? Curl 错误号

PHP curl_error函数

(PHP 4 >= 4.0.3, PHP 5)

curl_error — 返回一个保护当前会话最近一次错误的字符串

说明

```
string curl_error ( resource $ch )
```

返回一条最近一次cURL操作明确的文本的错误信息。

参数

ch

由 curl_init() 返回的 cURL 句柄。

返回值

返回错误信息或 '' (空字符串) 如果没有任何错误发生。

实例

```
<?php
// 创建一个指向一个不存在的位置的cURL句柄
$ch = curl_init('http://404.php.net/');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

if(curl_exec($ch) === false)
{
    echo 'Curl error: ' . curl_error($ch);
}
else
{
    echo '操作完成没有任何错误';
}

// 关闭句柄
curl_close($ch);
?>
```

参见

- curl_errno() - 返回最后一次的错误号
- ? Curl 错误代码

PHP curl_escape函数

(PHP 5 >= 5.5.0)

curl_escape — 对给定的字符串进行URL编码。

说明

```
string curl_escape ( resource $ch , string $str )
```

该函数对给定的字符串进行URL编码? RFC 3986。

参数

ch

由 curl_init() 返回的 cURL 句柄。

str

编码字符串

返回值

返回编码字符串，或者在失败时返回 FALSE。

实例

```
<?php
// 创建一个cURL句柄
$ch = curl_init();

// 编码GET参数
$location = curl_escape($ch, 'Hofbr?uhaus / München');
// Result: Hofbr%C3%A4uhaus%20%2F%20M%C3%BCnchen

// 比较编码后的URL
$url = "http://example.com/add_location.php?location={$location}";
// Result: http://example.com/add_location.php?location=Hofbr%C3%A4uhaus%20%2F%20M%C3%BCn

// 发送HTTP请求并关闭句柄
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_exec($ch);
curl_close($ch);
?>
```

PHP curl_exec函数

(PHP 4 >= 4.0.2, PHP 5)

curl_exec — 执行一个cURL会话

说明

```
mixed curl_exec ( resource $ch )
```

执行给定的cURL会话。

这个函数应该在初始化一个cURL会话并且全部的选项都被设置后被调用。

参数

ch

由 curl_init() 返回的 cURL 句柄。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。 然而，如果 CURLOPT_RETURNTRANSFER选项被设置，函数执行成功时会返回执行的结果，失败时返回 FALSE 。

实例

获取一个网页

```
<?php
// 创建一个cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.w3cschool.cc/");
curl_setopt($ch, CURLOPT_HEADER, 0);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close($ch);
?>
```

PHP curl_file_create函数

(PHP 5 >= 5.5.0)

curl_file_create — 创建一个 CURLFile 对象。

说明

```
CURLFile curl_file_create ( string $filename [, string $mimetype [, string $postname ]] )
```

创建一个 CURLFile 对象, 用与上传文件。

参数

filename

上传文件的路径

mimetype

文件的Mimetype

postname

文件名。

返回值

返回 CURLFile 对象。

实例

curl_file_create() 实例

```
<?php
/* http://example.com/upload.php:
<?php var_dump($_FILES); ?>
*/

// 创建一个 cURL 句柄
$ch = curl_init('http://example.com/upload.php');

// 创建一个 CURLFile 对象
$cfile = curl_file_create('cats.jpg','image/jpeg','test_name');

// 设置 POST 数据
$data = array('test_file' => $cfile);
curl_setopt($ch, CURLOPT_POST,1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);

// 执行句柄
curl_exec($ch);
?>
```

以上例程会输出：

```
array(1) {
  ["test_file"]=>
  array(5) {
    ["name"]=>
    string(9) "test_name"
    ["type"]=>
    string(10) "image/jpeg"
    ["tmp_name"]=>
    string(14) "/tmp/phpPC9Kbx"
    ["error"]=>
    int(0)
    ["size"]=>
    int(46334)
  }
}
```

PHP curl_getinfo函数

(PHP 4 >= 4.0.4, PHP 5)

curl_getinfo — 获取一个cURL连接资源句柄的信息

说明

```
mixed curl_getinfo ( resource $ch [, int $opt = 0 ] )
```

获取最后一次传输的相关信息。

参数

ch

由 curl_init() 返回的 cURL 句柄。

opt

这个参数可能是以下常量之一:

- **CURLINFO_EFFECTIVE_URL** - 最后一个有效的URL地址
- **CURLINFO_HTTP_CODE** - 最后一个收到的HTTP代码
- **CURLINFO_FILETIME** - 远程获取文档的时间，如果无法获取，则返回值为"-1"
- **CURLINFO_TOTAL_TIME** - 最后一次传输所消耗的时间
- **CURLINFO_NAMELOOKUP_TIME** - 名称解析所消耗的时间
- **CURLINFO_CONNECT_TIME** - 建立连接所消耗的时间
- **CURLINFO_PRETRANSFER_TIME** - 从建立连接到准备传输所使用的时间
- **CURLINFO_STARTTRANSFER_TIME** - 从建立连接到传输开始所使用的时间
- **CURLINFO_REDIRECT_TIME** - 在事务传输开始前重定向所使用的时间
- **CURLINFO_SIZE_UPLOAD** - 上传数据量的总值
- **CURLINFO_SIZE_DOWNLOAD** - 下载数据量的总值
- **CURLINFO_SPEED_DOWNLOAD** - 平均下载速度
- **CURLINFO_SPEED_UPLOAD** - 平均上传速度
- **CURLINFO_HEADER_SIZE** - header部分的大小
- **CURLINFO_HEADER_OUT** - 发送请求的字符串
- **CURLINFO_REQUEST_SIZE** - 在HTTP请求中有问题的请求的大小
- **CURLINFO_SSL_VERIFYRESULT** - 通过设置**CURLOPT_SSL_VERIFYPEER**返回的SSL证书验证请求的结果

- **CURLINFO_CONTENT_LENGTH_DOWNLOAD** - 从*Content-Length:* field中读取的下载内容长度
- **CURLINFO_CONTENT_LENGTH_UPLOAD** - 上传内容大小的说明
- **CURLINFO_CONTENT_TYPE** - 下载内容的*Content-Type:*值，NULL表示服务器没有发送有效的*Content-Type:* header

返回值

如果 opt 被设置，以字符串形式返回它的值。否则，返回返回一个包含下列元素的关联数组(它们分别对应于 opt):

- "url"
- "content_type"
- "http_code"
- "header_size"
- "request_size"
- "filetime"
- "ssl_verify_result"
- "redirect_count"
- "total_time"
- "namelookup_time"
- "connect_time"
- "pretransfer_time"
- "size_upload"
- "size_download"
- "speed_download"
- "speed_upload"
- "download_content_length"
- "upload_content_length"
- "starttransfer_time"
- "redirect_time"

更新日志

版本	说明
5.1.3	引入 `CURLINFO_HEADER_OUT` .

实例

```
<?php
// 创建一个cURL句柄
$ch = curl_init('http://www.yahoo.com/');

// 执行
curl_exec($ch);

// 检查是否有错误发生
if(!curl_errno($ch))
{
 $info = curl_getinfo($ch);

 echo 'Took ' . $info['total_time'] . ' seconds to send a request to ' . $info['url'];
}

// Close handle
curl_close($ch);
?>
```

PHP curl_init函数

(PHP 4 >= 4.0.2, PHP 5)

curl_init — 初始化一个cURL会话

说明

```
resource curl_init ([ string $url = NULL ] )
```

初始化一个新的会话，返回一个cURL句柄， 供curl_setopt(), curl_exec()和curl_close() 函数使用。

参数

url

如果提供了该参数，CURLOPT_URL 选项将会被设置成这个值。你也可以使用curl_setopt() 函数手动地设置这个值。

返回值

如果成功，返回一个cURL句柄，出错返回 FALSE。

实例

初始化一个新的cURL会话并获取一个网页

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.w3cschool.cc/");
curl_setopt($ch, CURLOPT_HEADER, 0);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close($ch);
?>
```

PHP curl_multi_add_handle函数

(PHP 5)

curl_multi_add_handle — 向curl批处理会话中添加单独的curl句柄

说明

```
int curl_multi_add_handle ( resource $mh , resource $ch )
```

增加 ch 句柄到批处理会话mh

参数

mh

由 curl_multi_init() 返回的 cURL 多个句柄。

ch

由 curl_init() 返回的 cURL 句柄。

返回值

成功时返回0，失败时返回CURLM_XXX之一的错误码。

实例

这个范例将会创建2个cURL句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建一对cURL资源
$ch1 = curl_init();
$ch2 = curl_init();

// 设置URL和相应的选项
curl_setopt($ch1, CURLOPT_URL, "http://www.w3cschool.cc/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加2个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$running=null;
// 执行批处理句柄
do {
    curl_multi_exec($mh,$running);
} while($running > 0);

// 关闭全部句柄
curl_multi_remove_handle($mh, $ch1);
curl_multi_remove_handle($mh, $ch2);
curl_multi_close($mh);
?>
```

PHP curl_multi_close函数

(PHP 5)

curl_multi_close — 关闭一组cURL句柄

说明

```
void curl_multi_close ( resource $mh )
```

关闭一组cURL句柄。

参数

mh

由 curl_multi_init() 返回的 cURL 多个句柄。

返回值

没有返回值。

实例

这个范例将会创建2个cURL句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建一对cURL资源
$ch1 = curl_init();
$ch2 = curl_init();

// 设置URL和相应的选项
curl_setopt($ch1, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加2个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$running=null;
// 执行批处理句柄
do {
    curl_multi_exec($mh,$running);
} while ($running > 0);

// 关闭全部句柄
curl_multi_remove_handle($mh, $ch1);
curl_multi_remove_handle($mh, $ch2);
curl_multi_close($mh);

?>
```

PHP curl_multi_exec函数

(PHP 5)

curl_multi_exec — 运行当前 cURL 句柄的子连接

说明

```
int curl_multi_exec ( resource $mh , int &$still_running )
```

处理在栈中的每一个句柄。无论该句柄需要读取或写入数据都可调用此方法。

参数

mh

由 curl_multi_init() 返回的 cURL 多个句柄。

still_running

一个用来判断操作是否仍在执行的标识的引用。

返回值

一个定义于 cURL 预定义常量中的 cURL 代码。

注意： 该函数仅返回关于整个批处理栈相关的错误。即使返回 CURLM_OK 时单个传输仍可能有问题。

实例

这个范例将会创建 2 个 cURL 句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建一对cURL资源
$ch1 = curl_init();
$ch2 = curl_init();

// 设置URL和相应的选项
curl_setopt($ch1, CURLOPT_URL, "http://lxr.php.net/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加2个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$active = null;
// 执行批处理句柄
do {
    $mrc = curl_multi_exec($mh, $active);
} while ($mrc == CURLM_CALL_MULTI_PERFORM);

while ($active && $mrc == CURLM_OK) {
    if (curl_multi_select($mh) != -1) {
        do {
            $mrc = curl_multi_exec($mh, $active);
        } while ($mrc == CURLM_CALL_MULTI_PERFORM);
    }
}

// 关闭全部句柄
curl_multi_remove_handle($mh, $ch1);
curl_multi_remove_handle($mh, $ch2);
curl_multi_close($mh);

?>
```

PHP curl_multi_getcontent函数

(PHP 5)

curl_multi_getcontent — 如果设置了CURLOPT_RETURNTRANSFER，则返回获取的输出的文本流

说明

```
string curl_multi_getcontent ( resource $ch )
```

如果CURLOPT_RETURNTRANSFER作为一个选项被设置到一个具体的句柄，那么这个函数将会以字符串的形式返回那个cURL句柄获取的内容。

参数

mh

由 curl_multi_init() 返回的 cURL 多个句柄。

返回值

如果设置了CURLOPT_RETURNTRANSFER，则返回获取的输出的文本流。

PHP curl_multi_info_read函数

(PHP 5)

curl_multi_info_read — 获取当前解析的cURL的相关传输信息

说明

```
array curl_multi_info_read ( resource $mh [, int &$msgs_in_queue = NULL ] )
```

查询批处理句柄是否单独的传输线程中有消息或信息返回。消息可能包含诸如从单独的传输线程返回的错误码或者只是传输线程有没有完成之类的报告。

重复调用这个函数，它每次都会返回一个新的结果，直到这时没有更多信息返回时，FALSE被当作一个信号返回。通过msgs_in_queue返回的整数指出将会包含当这次函数被调用后，还剩余的消息数。

注意: 返回的资源指向的数据调用curl_multi_remove_handle()后将不会存在。

参数

mh

由 curl_multi_init() 返回的 cURL 多个句柄。

msgs_in_queue

仍在队列中的消息数量。

返回值

成功时返回相关信息的数组，失败时返回FALSE。

返回值内容（返回数组的内容）：

键	值
msg	`CURLMSG_DONE` 常量。其他返回值当前不可用。
result	`CURLE_*` 常量之一。如果一切操作没有问题，将会返回 `CURLE_OK` 常量。
handle	cURL资源类型表明它有关的句柄。

实例

```
<?php

$urls = array(
   "http://www.baidu.com/",
   "http://www.google.com.hk/",
   "http://www.w3cschool.cc/"
);

$mh = curl_multi_init();

foreach ($urls as $i => $url) {
    $conn[$i] = curl_init($url);
    curl_setopt($conn[$i], CURLOPT_RETURNTRANSFER, 1);
    curl_multi_add_handle($mh, $conn[$i]);
}

do {
    $status = curl_multi_exec($mh, $active);
    $info = curl_multi_info_read($mh);
    if (false !== $info) {
        var_dump($info);
    }
} while ($status === CURLM_CALL_MULTI_PERFORM || $active);

foreach ($urls as $i => $url) {
    $res[$i] = curl_multi_getcontent($conn[$i]);
    curl_close($conn[$i]);
}

var_dump(curl_multi_info_read($mh));

?>
```

以上例程的输出类似于：

```
array(3) {
  ["msg"]=>
  int(1)
  ["result"]=>
  int(0)
  ["handle"]=>
  resource(5) of type (curl)
}
array(3) {
  ["msg"]=>
  int(1)
  ["result"]=>
  int(0)
  ["handle"]=>
  resource(7) of type (curl)
}
array(3) {
  ["msg"]=>
  int(1)
  ["result"]=>
  int(0)
  ["handle"]=>
  resource(6) of type (curl)
}
bool(false)
```

更新日志

版本	说明
5.2.0	`msgs_in_queue` 被加入。

PHP curl_multi_init函数

(PHP 5)

curl_multi_init — 返回一个新cURL批处理句柄

说明

```
resource curl_multi_init ( void )
```

允许并行地处理批处理cURL句柄。

参数

此函数没有参数。

返回值

成功时返回一个cURL批处理句柄，失败时返回FALSE。

实例

这个范例将会创建2个cURL句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建一对cURL资源
$ch1 = curl_init();
$ch2 = curl_init();

// 设置URL和相应的选项
curl_setopt($ch1, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加2个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$running=null;
// 执行批处理句柄
do {
    usleep(10000);
    curl_multi_exec($mh,$running);
} while ($running > 0);

// 关闭全部句柄
curl_multi_remove_handle($mh, $ch1);
curl_multi_remove_handle($mh, $ch2);
curl_multi_close($mh);

?>
```

PHP curl_multi_remove_handle函数

(PHP 5)

curl_multi_remove_handle — 移除curl批处理句柄资源中的某个句柄资源

说明

```
int curl_multi_remove_handle ( resource $mh , resource $ch )
```

从给定的批处理句柄mh中移除ch句柄。当ch句柄被移除以后，仍然可以合法地用curl_exec()执行这个句柄。当正在移除的句柄正在被使用，在处理的过程中所有的传输任务会被终止。

参数

mh

由 curl_multi_init() 返回的 cURL 多个句柄。

ch

由 curl_init() 返回的 cURL 句柄。

返回值

成功时返回一个cURL句柄，失败时返回FALSE。

PHP curl_multi_select函数

(PHP 5)

curl_multi_select — 等待所有cURL批处理中的活动连接

说明

```
int curl_multi_select ( resource $mh [, float $timeout = 1.0 ] )
```

阻塞直到cURL批处理连接中有活动连接。

参数

mh

由 curl_multi_init() 返回的 cURL 多个句柄。

timeout

以秒为单位，等待响应的时间。

返回值

成功时返回描述符集合中描述符的数量。失败时，select失败时返回-1，否则返回超时(从底层的select系统调用).

PHP curl_multi_setopt函数

(PHP 5 >= 5.5.0)

curl_multi_setopt — 设置一个批处理cURL传输选项。

说明

```
bool curl_multi_setopt ( resource $mh , int $option , mixed $value )
```

设置一个批处理cURL传输选项。

参数

ch

由 curl_init() 返回的 cURL 句柄。

option

需要设置的CURLOPT_XXX选项。

value

将设置在option选项上的值。

对于下面的这些option的可选参数，value应该被设置一个bool类型的值：

选项	可选*value*值
CURLOPT_AUTOREFERER	当根据*Location:*重定向时，自动设置header中 *Referer:*信息。
CURLOPT_BINARYTRANSFER	在启用**CURLOPT_RETURNTRANSFER**的时 返回原生的（Raw）输出。
CURLOPT_COOKIESESSION	启用时curl会仅仅传递一个session cookie，忽 他的cookie，默认状况下cURL会将所有的co 回给服务端。session cookie是指那些用来判 务器端的session是否有效而存在的cookie。
CURLOPT_CRLF	启用时将Unix的换行符转换成回车换行符。
CURLOPT_DNS_USE_GLOBAL_CACHE	启用时会启用一个全局的DNS缓存，此项为线 全的，并且默认启用。
	显示HTTP状态码，默认行为是忽略编号小于

CURLOPT_FAILONERROR	400的HTTP信息。
CURLOPT_FILETIME	启用时会尝试修改远程文档中的信息。结果信 通过curl*getinfo()*函数的*_CURLINFO_FILETI* 项返回。 curl_getinfo().
CURLOPT_FOLLOWLOCATION	启用时会将服务器服务器返回的*"Location: "*放 header中递归的返回给服务器，使 用**CURLOPT_MAXREDIRS**可以限定递归返 数量。
CURLOPT_FORBID_REUSE	在完成交互以后强迫断开连接，不能重用。
CURLOPT_FRESH_CONNECT	强制获取一个新的连接，替代缓存中的连接。
CURLOPT_FTP_USE_EPRT	启用时当FTP下载时，使用EPRT (或 LPRT) 设置为**FALSE**时禁用EPRT和LPRT，使用PO 令 only.
CURLOPT_FTP_USE_EPSV	启用时，在FTP传输过程中回复到PASV模式 先尝试EPSV命令。设置为**FALSE**时禁用EPS 令。
CURLOPT_FTPAPPEND	启用时追加写入文件而不是覆盖它。
CURLOPT_FTPASCII	**CURLOPT_TRANSFERTEXT**的别名。
CURLOPT_FTPLISTONLY	启用时只列出FTP目录的名字。
CURLOPT_HEADER	启用时会将头文件的信息作为数据流输出。
CURLINFO_HEADER_OUT	启用时追踪句柄的请求字符串。
CURLOPT_HTTPGET	启用时会设置HTTP的method为GET，因为G 默认是，所以只在被修改的情况下使用。
CURLOPT_HTTPPROXYTUNNEL	启用时会通过HTTP代理来传输。
CURLOPT_MUTE	启用时将cURL函数中所有修改过的参数恢复 值。
CURLOPT_NETRC	在连接建立以后，访问<var class="filename">~/.netrc</var>文件获取用户 密码信息连接远程站点。
CURLOPT_NOBODY	启用时将不对HTML中的BODY部分进行输出
CURLOPT_NOPROGRESS	启用时关闭curl传输的进度条，此项的默认设 启用。**Note**: PHP自动地设置这个选项为**TRU** 这个选项仅仅应当在以调试为目的时被改变。
CURLOPT_NOSIGNAL	启用时忽略所有的curl传递给php进行的信号。 SAPI多线程传输时此项被默认启用。
CURLOPT_POST	启用时会发送一个常规的POST请求，类型 为：*application/x-www-form-urlencoded*，就

	单提交的一样。
CURLOPT_PUT	启用时允许HTTP发送文件，必须同时设置**CURLOPT_INFILE**和**CURLOPT_INFILES**
CURLOPT_RETURNTRANSFER	将curl_exec()获取的信息以文件流的形式返回不是直接输出。
CURLOPT_SSL_VERIFYPEER	禁用后cURL将终止从服务端进行验证。使用**CURLOPT_CAINFO**选项设置证书使用**CURLOPT_CAPATH**选项设置证书目录 如**CURLOPT_SSL_VERIFYPEER**(默认值为2)用，**CURLOPT_SSL_VERIFYHOST**需要被成**TRUE**否则设置为**FALSE**。
CURLOPT_TRANSFERTEXT	启用后对FTP传输使用ASCII模式。对于LDA检索纯文本信息而非HTML。在Windows系统系统不会把*STDOUT*设置成binary模式。
CURLOPT_UNRESTRICTED_AUTH	在使用**CURLOPT_FOLLOWLOCATION**产生header中的多个locations中持续追加用户名信息，即使域名已发生改变。
CURLOPT_UPLOAD	启用后允许文件上传。
CURLOPT_VERBOSE	启用时会汇报所有的信息，存放在*STDERR*的**CURLOPT_STDERR**中。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

PHP curl_multi_strerror函数

(PHP 5 >= 5.5.0)

curl_multi_setopt — 返回描述错误码的字符串文本。

说明

```
string curl_multi_strerror ( int $errornum )
```

返回描述 CURLM 错误码的字符串文本。

参数

errornum

一个? CURLM error codes错误码常量。

返回值

返回描述错误码的字符串文本, 否则返回 NULL。

实例

```
<?php
// 创建 cURL 句柄
$ch1 = curl_init("http://www.w3cschool.cc/");
$ch2 = curl_init("http://php.net/");

// 创建一个批处理cURL句柄
$mh = curl_multi_init();

// 添加句柄到批处理句柄
curl_multi_add_handle($mh, $ch1);
curl_multi_add_handle($mh, $ch2);

// 执行批处理句柄
do {
    $status = curl_multi_exec($mh, $active);
    // 检查错误
    if($status > 0) {
        // 显示错误信息
        echo "ERROR!\n " . curl_multi_strerror($status);
    }
} while ($status === CURLM_CALL_MULTI_PERFORM || $active);
?>
```

PHP curl_pause函数

(PHP 5 >= 5.5.0)

curl_pause — 暂停及恢复连接。

说明

```
int curl_pause ( resource $ch , int $bitmask )
```

参数

ch

由 curl_init() 返回的 cURL 句柄。

bitmask

CURLPAUSE_*中的一个常量。

返回值

返回一个错误代码，如果没有错误返回 CURLE_OK 。

PHP curl_reset函数

(PHP 5 >= 5.5.0)

curl_reset— 重置libcurl会话句柄的所有选项。

说明

```
void curl_reset ( resource $ch )
```

该函数将重新初始化cURL的所有选项值（默认值）。

注意：curl_reset() 同样会重新设置 curl_init() 的 URL 参数。

参数

ch

由 curl_init() 返回的 cURL 句柄。

返回值

没有返回值。

实例

```
<?php
// 创建一个cURL句柄
$ch = curl_init();

// 设置 CURLOPT_USERAGENT 选项
curl_setopt($ch, CURLOPT_USERAGENT, "My test user-agent");

// 重置所有先前设置的选项
curl_reset($ch);

// 发送 HTTP 请求
curl_setopt($ch, CURLOPT_URL, 'http://w3cschool.cc/');
curl_exec($ch); // the previously set user-agent will be not sent, it has been reset by c

// 关闭句柄
curl_close($ch);
?>
```

PHP curl_setopt_array函数

(PHP 5 >= 5.1.3)

curl_setopt_array — 为cURL传输会话批量设置选项。

说明

```
bool curl_setopt_array ( resource $ch , array $options )
```

为cURL传输会话批量设置选项。这个函数对于需要设置大量的cURL选项是非常有用的，不需要重复地调用curl_setopt()。

参数

ch

由 curl_init() 返回的 cURL 句柄。

options

一个array用来确定将被设置的选项及其值。数组的键值必须是一个有效的curl_setopt()常量或者是它们对等的整数值。

返回值

如果全部的选项都被成功设置，返回TRUE。如果一个选项不能被成功设置，马上返回FALSE，忽略其后的任何在options数组中的选项。

实例

初始化一个新的cURL辉煌并抓取一个web页面。

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相应的选项
$options = array(CURLOPT_URL => 'http://www.w3cschool.cc/',
                 CURLOPT_HEADER => false
                );

curl_setopt_array($ch, $options);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close($ch);
?>
```

早于PHP 5.1.3这个函数可以做如下模拟：

我们对curl_setopt_array()的等价实现

```
<?php
if (!function_exists('curl_setopt_array')) {
   function curl_setopt_array(&$ch, $curl_options)
   {
       foreach ($curl_options as $option => $value) {
           if (!curl_setopt($ch, $option, $value)) {
               return false;
           }
       }
       return true;
   }
}
?>
```

注意：就curl_setopt()来说，传递一个数组到CURLOPT_POST将会把数据以multipart/form-data的方式编码，然而传递一个URL-encoded字符串将会以application/x-www-form-urlencoded的方式对数据进行编码。

PHP curl_setopt函数

(PHP 4 >= 4.0.2, PHP 5)

curl_setopt — 设置一个cURL传输选项。

说明

```
bool curl_setopt ( resource $ch , int $option , mixed $value )
```

为给定的cURL会话句柄设置一个选项。

参数

ch

由 curl_init() 返回的 cURL 句柄。

option

需要设置的CURLOPT_XXX选项。

value

将设置在option选项上的值。

对于下面的这些option的可选参数，value应该被设置一个bool类型的值：

选项	可选***value***值
CURLOPT_AUTOREFERER	当根据*Location:*重定向时，自动设置header中的*Referer:*信息。
CURLOPT_BINARYTRANSFER	在启用**CURLOPT_RETURNTRANSFER**的时候，返回原生的（Raw）输出。
CURLOPT_COOKIESESSION	启用时curl会仅仅传递一个session cookie，忽略其他的cookie，默认状况下cURL会将所有的cookie返回给服务端。session cookie是指那些用来判断服务器端的session是否有效而存在的cookie。
CURLOPT_CRLF	启用时将Unix的换行符转换成回车换行符。
CURLOPT_DNS_USE_GLOBAL_CACHE	启用时会启用一个全局的DNS缓存，此项为线程安全的，并且默认启用。
CURLOPT_FAILONERROR	显示HTTP状态码，默认行为是忽略编号小于

CURLOPT_FAILONERROR	400的HTTP信息。
CURLOPT_FILETIME	启用时会尝试修改远程文档中的信息。结果信 通过curl*getinfo()*函数的*_CURLINFO_FILETI* 项返回。 curl_getinfo().
CURLOPT_FOLLOWLOCATION	启用时会将服务器服务器返回的*"Location: "*放 header中递归的返回给服务器，使 用**CURLOPT_MAXREDIRS**可以限定递归返 数量。
CURLOPT_FORBID_REUSE	在完成交互以后强迫断开连接，不能重用。
CURLOPT_FRESH_CONNECT	强制获取一个新的连接，替代缓存中的连接。
CURLOPT_FTP_USE_EPRT	启用时当FTP下载时，使用EPRT (或 LPRT) 设置为**FALSE**时禁用EPRT和LPRT，使用PO 令 only.
CURLOPT_FTP_USE_EPSV	启用时，在FTP传输过程中回复到PASV模式 先尝试EPSV命令。设置为**FALSE**时禁用EPS 令。
CURLOPT_FTPAPPEND	启用时追加写入文件而不是覆盖它。
CURLOPT_FTPASCII	**CURLOPT_TRANSFERTEXT**的别名。
CURLOPT_FTPLISTONLY	启用时只列出FTP目录的名字。
CURLOPT_HEADER	启用时会将头文件的信息作为数据流输出。
CURLINFO_HEADER_OUT	启用时追踪句柄的请求字符串。
CURLOPT_HTTPGET	启用时会设置HTTP的method为GET，因为G 默认是，所以只在被修改的情况下使用。
CURLOPT_HTTPPROXYTUNNEL	启用时会通过HTTP代理来传输。
CURLOPT_MUTE	启用时将cURL函数中所有修改过的参数恢复 值。
CURLOPT_NETRC	在连接建立以后，访问<var class="filename">~/.netrc</var>文件获取用户 密码信息连接远程站点。
CURLOPT_NOBODY	启用时将不对HTML中的BODY部分进行输出
CURLOPT_NOPROGRESS	启用时关闭curl传输的进度条，此项的默认设 启用。**Note**: PHP自动地设置这个选项为**TRU** 这个选项仅仅应当在以调试为目的时被改变。
CURLOPT_NOSIGNAL	启用时忽略所有的curl传递给php进行的信号 SAPI多线程传输时此项被默认启用。
	启用时会发送一个常规的POST请求，类型

	单提交的一样。
CURLOPT_PUT	启用时允许HTTP发送文件，必须同时设置**CURLOPT_INFILE**和**CURLOPT_INFILES**
CURLOPT_RETURNTRANSFER	将curl_exec()获取的信息以文件流的形式返回不是直接输出。
CURLOPT_SSL_VERIFYPEER	禁用后cURL将终止从服务端进行验证。使用**CURLOPT_CAINFO**选项设置证书使用**CURLOPT_CAPATH**选项设置证书目录 如**CURLOPT_SSL_VERIFYPEER**(默认值为2)用，**CURLOPT_SSL_VERIFYHOST**需要被成**TRUE**否则设置为**FALSE**。
CURLOPT_TRANSFERTEXT	启用后对FTP传输使用ASCII模式。对于LDA检索纯文本信息而非HTML。在Windows系统系统不会把*STDOUT*设置成binary模式。
CURLOPT_UNRESTRICTED_AUTH	在使用**CURLOPT_FOLLOWLOCATION**产生header中的多个locations中持续追加用户名和信息，即使域名已发生改变。
CURLOPT_UPLOAD	启用后允许文件上传。
CURLOPT_VERBOSE	启用时会汇报所有的信息，存放在*STDERR*或的**CURLOPT_STDERR**中。

对于下面的这些option的可选参数，value应该被设置一个integer类型的值：

选项	可选*value*值
CURLOPT_BUFFERSIZE	每次获取的数据中读入缓存的大小，但是不保证这个值每次都会被填满。
CURLOPT_CLOSEPOLICY	不是CURLCLOSEPOLICY_LEAST_RECENTLY_USE就是CURLCLOSEPOLICY_OLDEST，还存在另三个CURLCLOSEPOLICY，但是cURL暂时还不支持。
CURLOPT_CONNECTTIMEOUT	在发起连接前等待的时间，如果设置为0，则无限等待。
CURLOPT_CONNECTTIMEOUT_MS	尝试连接等待的时间，以毫秒为单位。如果设置为0，则无限等待。

CURLOPT_DNS_CACHE_TIMEOUT	设置在内存中保存DNS信息的时间，默认为120秒。
CURLOPT_FTPSSLAUTH	FTP验证方式：*CURLFTPAUTH_SSL* (首先尝试SSL)，*CURLFTPAUTH_TLS* (首先尝试TLS)或*CURLFTPAUTH_DEFAULT* (让cURL自动决定)
CURLOPT_HTTP_VERSION	*CURL_HTTP_VERSION_NONE* (默认值，让cUR自己判断使用哪个版本)，*CURL_HTTP_VERSION_1_0* (强制使用 HTTP/1.0)或*CURL_HTTP_VERSION_1_1* (强制使用 HTTP/1.1)。
CURLOPT_INFILESIZE	设定上传文件的大小限制，字节(byte)为单位。
CURLOPT_LOW_SPEED_LIMIT	当传输速度小于**CURLOPT_LOW_SPEED_LIMIT**时(bytes/sec)，PHP会根据**CURLOPT_LOW_SPEED_TIME**来判断是否因慢而取消传输。
CURLOPT_LOW_SPEED_TIME	当传输速度小于**CURLOPT_LOW_SPEED_LIMIT**时(bytes/sec)，PHP会根据**CURLOPT_LOW_SPEED_TIME**来判断是否因慢而取消传输。
CURLOPT_MAXCONNECTS	允许的最大连接数量，超过是会通过**CURLOPT_CLOSEPOLICY**决定应该停止哪些连接。
CURLOPT_MAXREDIRS	指定最多的HTTP重定向的数量，这个选项是和**CURLOPT_FOLLOWLOCATION**一起使用的。
CURLOPT_PORT	用来指定连接端口。（可选项）
CURLOPT_PROTOCOLS	**CURLPROTO_***的位域指。如果被启用，位域值会限定libcurl在传输过程中有哪些可使用的协议。这允许你在编译libcurl时支持众多协议，但是限制只用它们中被允许使用的一个子集。默认libcurl将会用全部它支持的协议。参见**CURLOPT_REDIR_PROTOCOLS**.可用的协议选项为：CURLPROTO_HTTP、CURLPROTO_HTTPS、CURLPROTO_FTP、CURLPROTO_FTPS、CURLPROTO_SCP、CURLPROTO_SFTP、CURLPROTO_TELNET、CURLPROTO_LDAP、CURLPROTO_LDAPS、CURLPROTO_DICT、CURLPROTO_FILE、CURLPROTO_TFTP、CURLPROTO_ALL

CURLOPT_PROTOCOLS	**CURLPROTO_***的位域指。如果被启用，位域值会限定libcurl在传输过程中有哪些可使用的协议。这允许你在编译libcurl时支持众多协议，但是限制只用它们中被允许使用的一个子集。默认libcurl将会用全部它支持的协议。参见**CURLOPT_REDIR_PROTOCOLS**.可用的协议选项为：CURLPROTO_HTTP、CURLPROTO_HTTPS、CURLPROTO_FTP、CURLPROTO_FTPS、CURLPROTO_SCP、CURLPROTO_SFTP、CURLPROTO_TELNET、CURLPROTO_LDAP、CURLPROTO_LDAPS、CURLPROTO_DICT、CURLPROTO_FILE、CURLPROTO_TFTP、CURLPROTO_ALL
CURLOPT_PROXYAUTH	HTTP代理连接的验证方式。使用在**CURLOPT_HTTPAUTH**中的位域标志来设置相应选项。对于代理验证只有*CURLAUTH_BASIC*和*CURLAUTH_NTLM*当前被支持。
CURLOPT_PROXYPORT	代理服务器的端口。端口也可以在**CURLOPT_PROXY**中进行设置。
CURLOPT_PROXYTYPE	不是*CURLPROXY_HTTP* (默认值) 就是*CURLPROXY_SOCKS5*。
CURLOPT_REDIR_PROTOCOLS	**CURLPROTO_***中的位域值。如果被启用，位域值将会限制传输线程在**CURLOPT_FOLLOWLOCATION**开启时跟随某个重定向时可使用的协议。这将使你对重定向时限制传输线程使用被允许的协议子集默认libcurl将会允许除FILE和SCP之外的全部协议。这个和7.19.4预发布版本种无条件地跟随所有支持的协议有一些不同。关于协议常量，请参照**CURLOPT_PROTOCOLS**。
CURLOPT_RESUME_FROM	在恢复传输时传递一个字节偏移量（用来断点续传）。
CURLOPT_SSL_VERIFYHOST	1 检查服务器SSL证书中是否存在一个公用名(common name)。译者注：公用名(Common Name)一般来讲就是填写你将要申请SSL证书的域名 (domain)或子域名(sub domain)。2 检查公用名是否存在，并且是否与提供的主机名匹配。
CURLOPT_SSLVERSION	使用的SSL版本(2 或 3)。默认情况下PHP会自己检测这个值，尽管有些情况下需要手动地进行设置。
	如果在**CURLOPT_TIMEVALUE**指定的某个时间以后被编辑过，则使

CURLOPT_TIMECONDITION	用*CURL_TIMECOND_IFMODSINCE*返回页面，果没有被修改过，并且**CURLOPT_HEADER**为true，则返回一个*"304 Not Modified"*的header，**CURLOPT_HEADER**为false，则使用*CURL_TIMECOND_IFUNMODSINCE*，默认值为*CURL_TIMECOND_IFUNMODSINCE*。
CURLOPT_TIMEOUT	设置cURL允许执行的最长秒数。
CURLOPT_TIMEOUT_MS	设置cURL允许执行的最长毫秒数。
CURLOPT_TIMEVALUE	设置一个**CURLOPT_TIMECONDITION**使用的时戳，在默认状态下使用的是*CURL_TIMECOND_IFMODSINCE*。

对于下面的这些option的可选参数，value应该被设置一个string类型的值：

选项	可选***value***值
CURLOPT_CAINFO	一个保存着1个或多个用来让服务端验证的证书的文件名。这个参数仅仅在和**CURLOPT_SSL_VERIFYPEER**一起使用时才有意义。 .
CURLOPT_CAPATH	一个保存着多个CA证书的目录。这个选项是和**CURLOPT_SSL_VERIFYPEER**一起使用的。
CURLOPT_COOKIE	设定HTTP请求中*"Cookie: "*部分的内容。多个cookie用分号分隔，分号后带一个空格(例如，"*fruit=apple; colour=red*")。
CURLOPT_COOKIEFILE	包含cookie数据的文件名，cookie文件的格式可以是Netscape格式，或者只是纯HTTP头部信息存入文件。
CURLOPT_COOKIEJAR	连接结束后保存cookie信息的文件。
CURLOPT_CUSTOMREQUEST	使用一个自定义的请求信息来代替*"GET"*或*"HEAD"*作为HTTP请求。这对于执行*"DELETE"* 或者其他更隐蔽的HTTP请求。有效值如*"GET"*，*"POST"*，*"CONNECT"*等等。也就是说，不要在这里输入整个HTTP请求。例如输入*"GET /index.html HTTP/1.0\r\n\r\n"*是不正确的。**Note**: 在确定服务器支持这个自定义请求的

	方法前不要使用。
CURLOPT_EGDSOCKET	类似**CURLOPT_RANDOM_FILE**，除了一个Entropy Gathering Daemon套接字。
CURLOPT_ENCODING	HTTP请求头中*"Accept-Encoding: "*的值。支持的编码有*"identity"*，*"deflate"*和*"gzip"*。如果为空字符串*""*，请求头会发送所有支持的编码类型。
CURLOPT_FTPPORT	这个值将被用来获取供FTP"POST"指令所需要的IP地址。"POST"指令告诉远程服务器连接到我们指定的IP地址。这个字符串可以是纯文本的IP地址、主机名、一个网络接口名（UNIX下）或者只是一个'-'来使用默认的IP地址。
CURLOPT_INTERFACE	网络发送接口名，可以是一个接口名、IP地址或者是一个主机名。
CURLOPT_KRB4LEVEL	KRB4 (Kerberos 4) 安全级别。下面的任何值都是有效的(从低到高的顺序)：*"clear"*、*"safe"*、*"confidential"*、*"private"*.。如果字符串和这些都不匹配，将使用*"private"*。这个选项设置为**NULL**时将禁用KRB4 安全认证。目前KRB4 安全认证只能用于FTP传输。
CURLOPT_POSTFIELDS	全部数据使用HTTP协议中的"POST"操作来发送。要发送文件，在文件名前面加上@前缀并使用完整路径。这个参数可以通过urlencoded后的字符串类似'*para1=val1&para2=val2&...*'或使用一个以字段名为键值，字段数据为值的数组。如果*value*是一个数组，*Content-Type*头将会被设置成*multipart/form-data*。
CURLOPT_PROXY	HTTP代理通道。
CURLOPT_PROXYUSERPWD	一个用来连接到代理的*"[username]:[password]"*格式的字符串。
CURLOPT_RANDOM_FILE	一个被用来生成SSL随机数种子的文件名。
CURLOPT_RANGE	以*"X-Y"*的形式，其中X和Y都是可选项获取数据的范围，以字节计。HTTP传输线程也支持几个这样的重复项中间用逗号分隔如*"X-Y,N-M"*。
CURLOPT_REFERER	在HTTP请求头中*"Referer: "*的内容。
CURLOPT_SSL_CIPHER_LIST	一个SSL的加密算法列表。例如*RC4-SHA*和*TLSv1*都是可用的加密列表。
CURLOPT_SSLCERT	一个包含PEM格式证书的文件名。
CURLOPT_SSLCERTPASSWD	使用**CURLOPT_SSLCERT**证书需要的密码。

CURLOPT_SSLCERTTYPE	证书的类型。支持的格式有*"PEM"* (默认值), *"DER"*和*"ENG"*。
CURLOPT_SSLENGINE	用来在**CURLOPT_SSLKEY**中指定的SSL私钥的加密引擎变量。
CURLOPT_SSLENGINE_DEFAULT	用来做非对称加密操作的变量。
CURLOPT_SSLKEY	包含SSL私钥的文件名。
CURLOPT_SSLKEYPASSWD	在**CURLOPT_SSLKEY**中指定了的SSL私钥的密码。**Note**: 由于这个选项包含了敏感的密码信息，记得保证这个PHP脚本的安全。
CURLOPT_SSLKEYTYPE	**CURLOPT_SSLKEY**中规定的私钥的加密类型，支持的密钥类型为*"PEM"*(默认值)、*"DER"*和*"ENG"*。
CURLOPT_URL	需要获取的URL地址，也可以在curl_init()函数中设置。
CURLOPT_USERAGENT	在HTTP请求中包含一个*"User-Agent: "*头的字符串。
CURLOPT_USERPWD	传递一个连接中需要的用户名和密码，格式为：*"[username]:[password]"*。

对于下面的这些option的可选参数，value应该被设置一个数组：

选项	可选***value***值	备注
CURLOPT_HTTP200ALIASES	200响应码数组，数组中的响应吗被认为是正确的响应，否则被认为是错误的。	在cURL 7.10.3中被加入。
CURLOPT_HTTPHEADER	一个用来设置HTTP头字段的数组。使用如下的形式的数组进行设置：array('Content-type: text/plain', 'Content-length: 100')	
CURLOPT_POSTQUOTE	在FTP请求执行完成后，在服务器上执行的一组FTP命令。	
CURLOPT_QUOTE	一组先于FTP请求的在服务器上执行的FTP命令。	

对于下面的这些option的可选参数，value应该被设置一个流资源 （例如使用fopen()）：

选项	可选*value*值
CURLOPT_FILE	设置输出文件的位置，值是一个资源类型，默认为*STDOUT* (浏览器)。
CURLOPT_INFILE	在上传文件的时候需要读取的文件地址，值是一个资源类型。
CURLOPT_STDERR	设置一个错误输出地址，值是一个资源类型，取代默认的*STDERR*。
CURLOPT_WRITEHEADER	设置header部分内容的写入的文件地址，值是一个资源类型。

对于下面的这些option的可选参数，value应该被设置为一个回调函数名：

选项	可选*value*值
CURLOPT_HEADERFUNCTION	设置一个回调函数，这个函数有两个参数，第一个是cURL的资源句柄，第二个是输出的header数据。header数据的输出必须依赖这个函数，返回已写入的数据大小。
CURLOPT_PASSWDFUNCTION	设置一个回调函数，有三个参数，第一个是cURL的资源句柄，第二个是一个密码提示符，第三个参数是密码长度允许的最大值。返回密码的值。
CURLOPT_PROGRESSFUNCTION	设置一个回调函数，有三个参数，第一个是cURL的资源句柄，第二个是一个文件描述符资源，第三个是长度。返回包含的数据。
CURLOPT_READFUNCTION	回调函数名。该函数应接受三个参数。第一个是 cURL resource；第二个是通过选项 **CURLOPT_INFILE** 传给 cURL 的 stream resource；第三个参数是最大可以读取的数据的数量。回 调函数必须返回一个字符串，长度小于或等于请求的数据量（第三个参数）。一般从传入的 stream resource 读取。返回空字符串作为 *EOF*（文件结束） 信号。
CURLOPT_WRITEFUNCTION	回调函数名。该函数应接受两个参数。第一个是 cURL resource；第二个是要写入的数据字符串。数 据必须在函数中被保存。函数必须返回准确的传入的要写入数据的字节数，否则传输会被一个错误所中 断。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

更新日志

版本	说明
5.2.10	引入 `CURLOPT_PROTOCOLS` , and `CURLOPT_REDIR_PROTOCOLS` .
5.1.0	引入 `CURLOPT_AUTOREFERER` , `CURLOPT_BINARYTRANSFER` , `CURLOPT_FTPSSLAUTH` , `CURLOPT_PROXYAUTH` , and `CURLOPT_TIMECONDITION` .
5.0.0	引入 `CURLOPT_FTP_USE_EPRT` , `CURLOPT_NOSIGNAL` , `CURLOPT_UNRESTRICTED_AUTH` , `CURLOPT_BUFFERSIZE` , `CURLOPT_HTTPAUTH` , `CURLOPT_PROXYPORT` , `CURLOPT_PROXYTYPE` , `CURLOPT_SSLCERTTYPE` , and `CURLOPT_HTTP200ALIASES` .

实例

初始化一个新的cURL会话并获取一个网页

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch, CURLOPT_HEADER, false);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

//关闭cURL资源，并且释放系统资源
curl_close($ch);
?>
```

上传文件实例:

```
<?php

/* http://localhost/upload.php:
print_r($_POST);
print_r($_FILES);
*/

$ch = curl_init();

$data = array('name' => 'Foo', 'file' => '@/home/user/test.png');

curl_setopt($ch, CURLOPT_URL, 'http://localhost/upload.php');
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);

curl_exec($ch);
?>
```

以上实例输出结果如下：

```
Array
(
    [name] => Foo
)
Array
(
    [file] => Array
        (
            [name] => test.png
            [type] => image/png
            [tmp_name] => /tmp/phpcpjNeQ
            [error] => 0
            [size] => 279
        )

)
```

注释

传递一个数组到CURLOPT_POSTFIELDS，cURL会把数据编码成 multipart/form-data，而然传递一个URL-encoded字符串时，数据会被编码成 application/x-www-form-urlencoded。

PHP curl_share_close函数

(PHP 5 >= 5.5.0)

curl_share_close — 关闭 cURL 共享句柄

说明

```
void curl_share_close ( resource $sh )
```

关闭 cURL 共享句柄并释放所有资源。

参数

sh

通过curl_share_init()返回cURL共享句柄。

返回值

没有返回值。

实例

该实例将创建一个cURL共享句柄，并添加两个 cURL 句柄，两个句柄共享cookie数据。

```
<?php
// 创建cURL共享句柄并设置cookie数据
$sh = curl_share_init();
curl_share_setopt($sh, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);

// 初始化第一个cURL句柄并指定它为共享句柄
$ch1 = curl_init("http://www.w3cschool.cc/");
curl_setopt($ch1, CURLOPT_SHARE, $sh);

// 执行第一个cURL句柄
curl_exec($ch1);

// 初始化第二个cURL句柄并指定它为共享句柄
$ch2 = curl_init("http://php.net/");
curl_setopt($ch2, CURLOPT_SHARE, $sh);

// 执行第二个cURL句柄
//  所有 $ch1 句柄的数据在 $ch2 句柄中共享
curl_exec($ch2);

// 关闭cURL共享句柄
curl_share_close($sh);

// 关闭cURL句柄
curl_close($ch1);
curl_close($ch2);
?>
```

PHP curl_share_init函数

(PHP 5 >= 5.5.0)

curl_share_init — 初始化一个 cURL 共享句柄

说明

```
resource curl_share_init ( void )
```

允许两个 cURL 句柄分享数据。

参数

此函数没有参数。

返回值

返回"cURL共享句柄"的资源。

实例

该实例将创建一个cURL共享句柄，并添加两个 cURL 句柄，两个句柄共享cookie数据。

```
<?php
// 创建cURL共享句柄并设置cookie数据
$sh = curl_share_init();
curl_share_setopt($sh, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);

// 初始化第一个cURL句柄并指定它为共享句柄
$ch1 = curl_init("http://www.w3cschool.cc/");
curl_setopt($ch1, CURLOPT_SHARE, $sh);

// 执行第一个cURL句柄
curl_exec($ch1);

// 初始化第二个cURL句柄并指定它为共享句柄
$ch2 = curl_init("http://php.net/");
curl_setopt($ch2, CURLOPT_SHARE, $sh);

// 执行第二个cURL句柄
//  所有 $ch1 句柄的数据在 $ch2 句柄中共享
curl_exec($ch2);

// 关闭cURL共享句柄
curl_share_close($sh);

// 关闭cURL句柄
curl_close($ch1);
curl_close($ch2);
?>
```

PHP curl_share_setopt函数

(PHP 5 >= 5.5.0)

curl_share_setopt — 设置 cURL 共享句柄的一个选项。

说明

```
bool curl_share_setopt ( resource $sh , int $option , string $value )
```

设置 cURL 共享句柄的一个选项。

参数

sh

通过 curl_share_init() 初始化的共享句柄。

option

选项	描述
`CURLSHOPT_SHARE`	指定共享的数据类型
`CURLSHOPT_UNSHARE`	指定不共享的数据类型

value

值	描述
`CURL_LOCK_DATA_COOKIE`	共享cookie数据
`CURL_LOCK_DATA_DNS`	共享 DNS 缓存。
`CURL_LOCK_DATA_SSL_SESSION`	共享 SSL session ID, 减少连接到相同的服务器花费在 SSL 握手时的时间。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

该实例将创建一个cURL共享句柄，并添加两个 cURL 句柄，两个句柄共享cookie数据。

```
<?php
// 创建cURL共享句柄并设置cookie数据
$sh = curl_share_init();
curl_share_setopt($sh, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);

// 初始化第一个cURL句柄并指定它为共享句柄
$ch1 = curl_init("http://www.w3cschool.cc/");
curl_setopt($ch1, CURLOPT_SHARE, $sh);

// 执行第一个cURL句柄
curl_exec($ch1);

// 初始化第二个cURL句柄并指定它为共享句柄
$ch2 = curl_init("http://php.net/");
curl_setopt($ch2, CURLOPT_SHARE, $sh);

// 执行第二个cURL句柄
//  所有 $ch1 句柄的数据在 $ch2 句柄中共享
curl_exec($ch2);

// 关闭cURL共享句柄
curl_share_close($sh);

// 关闭cURL句柄
curl_close($ch1);
curl_close($ch2);
?>
```

PHP curl_strerror函数

(PHP 5 >= 5.5.0)

curl_strerror — 返回错误码的描述。

说明

```
string curl_strerror ( int $errornum )
```

返回错误码的文本描述信息。

参数

errornum

一个 ? cURL 错误码的常量。

返回值

返回错误码描述信息，非法错误码返回NULL 。

实例

```
<?php
// 创建一个拼写错误的URL的cURL句柄
$ch = curl_init("htp://example.com/");

// 发送请求
curl_exec($ch);

// 检查错误代码并显示错误信息
if($errno = curl_errno($ch)) {
    $error_message = curl_strerror($errno);
    echo "cURL error ({$errno}):\n {$error_message}";
}

// 关闭句柄
curl_close($ch);
?>
```

以上例程会输出：

```
cURL error (1):
 Unsupported protocol
```

PHP curl_unescape函数

(PHP 5 >= 5.5.0)

curl_unescape — 解码经过URL编码的字符串。

说明

```
string curl_unescape ( resource $ch , string $str )
```

解码经过URL编码的字符串。

注意： curl_unescape() 不能解码加号 (+) 为空格, urldecode() 可以。

参数

ch

由 curl_init() 返回的 cURL 句柄。

str

URL 编码的字符串

返回值

返回解码字符串或者在失败时返回 FALSE。

实例

```
<?php
// 创建一个curl句柄
$ch = curl_init('http://example.com/redirect.php');

// 发送 HTTP 请求
curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 1);
curl_exec($ch);

// 获得最后一个有效的URL
$effective_url = curl_getinfo($ch, CURLINFO_EFFECTIVE_URL);
// ie. "http://example.com/show_location.php?loc=M%C3%BCnchen"

// 解码URL
$effective_url_decoded = curl_unescape($ch, $effective_url);
// "http://example.com/show_location.php?loc=München"

// 关闭句柄
curl_close($ch);
?>
```

PHP curl_version函数

(PHP 5 >= 5.5.0)

curl_version — 获取cURL版本信息。

说明

```
array curl_version ([ int $age = CURLVERSION_NOW ] )
```

返回关于cURL的版本信息。

参数

age

返回值

返回一个相关的数组包含如下元素：

Indice	值描述
version_number	cURL 24位版本号
version	cURL 版本号，字符串形式
ssl_version_number	OpenSSL 24 位版本号
ssl_version	OpenSSL 版本号，字符串形式
libz_version	zlib 版本号，字符串形式
host	关于编译cURL主机的信息
age	
features	一个*CURL_VERSION_XXX*常量的位掩码
protocols	一个cURL支持的协议名字的数组

实例

这个范例将会检查当前cURL版本使用curl_version()返回的'features'位掩码中哪些特性是可用的。

```
<?php
// 获取cURL版本数组
$version = curl_version();

// 在cURL编译版本中使用位域来检查某些特性
$bitfields = Array(
            'CURL_VERSION_IPV6',
            'CURL_VERSION_KERBEROS4',
            'CURL_VERSION_SSL',
            'CURL_VERSION_LIBZ'
            );

foreach($bitfields as $feature)
{
    echo $feature . ($version['features'] & constant($feature) ? ' matches' : ' does not
    echo PHP_EOL;
}
?>
```

PHP Date / Time 函数

PHP Date / Time 简介

date/time 函数允许您提取并格式化服务器上的日期和时间。

注释：这些函数依赖于服务器的本地设置。

安装

date/time 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

Runtime 配置

日期/时间函数的行为受到 php.ini 中设置的影响。

Date/Time 配置选项：

名称	默认	描述	可改变
date.default_latitude	"31.7667"	规定默认纬度（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.default_longitude	"35.2333"	规定默认经度（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.sunrise_zenith	"90.83"	规定日出天顶（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.sunset_zenith	"90.83"	规定日落天顶（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.timezone	""	规定默认时区（从 PHP 5.1 开始可用）。	PHP_INI_ALL

PHP Date / Time 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
checkdate()	验证格利高里日期。	3
date_default_timezone_get()	返回默认时区。	5
date_default_timezone_set()	设置默认时区。	5
date_sunrise()	返回给定的日期与地点的日出时间。	5
date_sunset()	返回给定的日期与地点的日落时间。	5
date()	格式化本地时间／日期。	3
getdate()	返回日期／时间信息。	3
gettimeofday()	返回当前时间信息。	3
gmdate()	格式化 GMT/UTC 日期/时间。	3
gmmktime()	取得 GMT 日期的 UNIX 时间戳。	3
gmstrftime()	根据本地区域设置格式化 GMT/UTC 时间／日期。	3
idate()	将本地时间/日期格式化为整数	5
localtime()	返回本地时间。	4
microtime()	返回当前时间的微秒数。	3
mktime()	返回一个日期的 Unix 时间戳。	3
strftime()	根据区域设置格式化本地时间／日期。	3
strptime()	解析由 strftime 生成的日期／时间。	5
strtotime()	将任何英文文本的日期或时间描述解析为 Unix 时间戳。	3
time()	返回当前时间的 Unix 时间戳。	3

PHP Date / Time 常量

PHP：指示支持该常量的最早的 PHP 版本。

常量	描述	PHP
DATE_ATOM	原子钟格式 (如: 2005-08-15T16:13:03+0000)	
DATE_COOKIE	HTTP Cookies 格式 (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_ISO8601	ISO-8601 (如: 2005-08-14T16:13:03+0000)	
DATE_RFC822	RFC 822 (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_RFC850	RFC 850 (如: Sunday, 14-Aug-05 16:13:03 UTC)	
DATE_RFC1036	RFC 1036 (如: Sunday, 14-Aug-05 16:13:03 UTC)	
DATE_RFC1123	RFC 1123 (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_RFC2822	RFC 2822 (如: Sun, 14 Aug 2005 16:13:03 +0000)	
DATE_RSS	RSS (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_W3C	World Wide Web Consortium (如: 2005-08-14T16:13:03+0000)	

PHP checkdate() 函数

定义和用法

checkdate() 函数验证一个格里高里日期。

如果指定的值合法，则该函数返回 true，否则返回 false。

日期在下列情况下为合法：

- month 介于且包括 1 - 12
- Day 的值在给定的 month 所应该具有的天数范围之内，闰年已经考虑进去了。
- year 介于且包括 1 到 32767

语法

```
checkdate(month,day,year)
```

参数	描述
month	必需。规定月。
day	必需。规定日。
year	必需。规定年。

例子

```
<?php
var_dump(checkdate(12,31,2000));
var_dump(checkdate(2,29,2003));
var_dump(checkdate(2,29,2004));
?>
```

输出：

```
bool(true)
bool(false)
bool(true)
```

PHP date_default_timezone_get() 函数

定义和用法

date_default_timezone_get() 函数返回脚本中所有日期时间函数所使用的默认时区。

语法

```
date_default_timezone_get(void)
```

参数	描述
void	可选。

说明

本函数返回默认时区，使用如下“假定”的顺序：

- 用 date_default_timezone_set() 函数设定的时区（如果设定了的话）
- TZ 环境变量（如果非空）
- date.timezone 配置选项（如果设定了的话）
- 自己推测（如果操作系统支持）
- 如果以上选择都不成功，则返回 UTC

例子

```
<?php
echo(date_default_timezone_get());
?>
```

输出：

```
Europe/Paris
```

PHP date_default_timezone_set() 函数

定义和用法

date_default_timezone_set() 函数设置用在脚本中所有日期/时间函数的默认时区。

语法

```
date_default_timezone_set(timezone)
```

参数	描述
timezone	必需。时区标识符，比如 "UTC" 或 "Europe/Paris"。合法时区的列表：http://www.php.net/manual/en/timezones.php

说明

注释：自 PHP 5.1.0 起（此版本日期时间函数被重写了），如果时区不合法则每个对日期时间函数的调用都会产生一条 E_NOTICE 级别的错误信息，如果使用系统设定或 TZ 环境变量则还会产生 E_STRICT 级别的信息。

例子

```
<?php
echo(date_default_timezone_set("Europe/Paris"));
?>
```

输出：

```
1
```

PHP date_sunrise() 函数

定义和用法

date_sunrise() 函数返回指定的日期与地点的日出时间。

语法

```
date_sunrise(timestamp,format,latitude,longitude,zenith,gmt_offset)
```

参数	描述
timestamp	必需。
format	可选。规定如何返回结果： SUNFUNCS_RET_STRING （以 string 格式返回结果，比如 16:46） SUNFUNCS_RET_DOUBLE （以 float 格式返回结果，比如 16.78243132） SUNFUNCS_RET_TIMESTAMP （以 integer 格式（时间戳）返回结果，比如 1095034606）
latitude	可选。规定地点的纬度。默认是指北纬。因此如果要指定南纬，必须传递一个负值。
longitude	可选。规定地点的经度。默认是指东经。因此如果要指定西经，必须传递一个负值。
zenith	可选。
gmt_offset	可选。规定 GMT 与本地时间的差值。单位是小时。

例子

```
<?php
//计算葡萄牙里斯本的日出时间
//Latitude: 北纬 38.4 度
//Longitude: 西经 9 度
//Zenith ~= 90
//offset: +1 GMT
echo("Date: " . date("D M d Y") . "<br />");
echo("Sunrise time: ");
echo(date_sunrise(time(),SUNFUNCS_RET_STRING,38.4,-9,90,1));
?>
```

输出：

```
Date: Tue Jan 24 2006
Sunrise time: 08:52
```

PHP date_sunset() 函数

定义和用法

date_sunset() 函数返回指定的日期与地点的日落时间。

语法

```
date_sunset(timestamp,format,latitude,longitude,zenith,gmt_offset)
```

参数	描述
timestamp	必需。
format	可选。规定如何返回结果： SUNFUNCS_RET_STRING (以 string 格式返回结果，比如 16:46) SUNFUNCS_RET_DOUBLE (以 float 格式返回结果，比如 16.78243132) SUNFUNCS_RET_TIMESTAMP (以 integer 格式（时间戳）返回结果，比如 1095034606)
latitude	可选。规定地点的纬度。默认是指北纬。因此如果要指定南纬，必须传递一个负值。
longitude	可选。规定地点的经度。默认是指东经。因此如果要指定西经，必须传递一个负值。
zenith	可选。
gmt_offset	可选。规定 GMT 与本地时间的差值。单位是小时。

例子

```
<?php
//计算葡萄牙里斯本的日出时间
//Latitude: 北纬 38.4 度
//Longitude: 西经 9 度
//Zenith ~= 90
//offset: +1 GMT
echo("Date: " . date("D M d Y") . "<br />");
echo("Sunrise time: ");
echo(date_sunset(time(),SUNFUNCS_RET_STRING,38.4,-9,90,1));
?>
```

输出：

```
Date: Tue Jan 24 2006
Sunrise time: 18:44
```

PHP date() 函数

定义和用法

date() 函数格式化一个本地时间／日期。

语法

```
date(format,timestamp)
```

参数	描述
format	必需。规定如何返回结果。
timestamp	可选。

例子

```
<?php
echo("Result with date():<br />");
echo(date("l") . "<br />");
echo(date("l dS \of F Y h:i:s A") . "<br />");
echo("Oct 3,1975 was on a ".date("l", mktime(0,0,0,10,3,1975))."<br />");
echo(date(DATE_RFC822) . "<br />");
echo(date(DATE_ATOM,mktime(0,0,0,10,3,1975)) . "<br /><br />");

echo("Result with gmdate():<br />");
echo(gmdate("l") . "<br />");
echo(gmdate("l dS \of F Y h:i:s A") . "<br />");
echo("Oct 3,1975 was on a ".gmdate("l", mktime(0,0,0,10,3,1975))."<br />");
echo(gmdate(DATE_RFC822) . "<br />");
echo(gmdate(DATE_ATOM,mktime(0,0,0,10,3,1975)) . "<br />");
?>
```

输出：

```
Result with date():
Tuesday
Tuesday 24th of January 2006 02:41:22 PM
Oct 3,1975 was on a Friday
Tue, 24 Jan 2006 14:41:22 CET
1975-10-03T00:00:00+0100

Result with gmdate():
Tuesday
Tuesday 24th of January 2006 01:41:22 PM
Oct 3,1975 was on a Thursday
Tue, 24 Jan 2006 13:41:22 GMT
1975-10-02T23:00:00+0000
```

PHP getdate() 函数

定义和用法

getdate() 函数取得日期／时间信息。

语法

```
getdate(timestamp)
```

参数	描述
timestamp	可选。规定 Unix 时间格式中的时间。

说明

返回一个根据 *timestamp* 得出的包含有日期信息的结合数组。如果没有给出时间戳，则认为是当前本地时间。

数组中的单元如下：

键名	说明	返回值例子
"seconds"	秒的数字表示	0 到 59
"minutes"	分钟的数字表示	0 到 59
"hours"	小时的数字表示	0 到 23
"mday"	月份中第几天的数字表示	1 到 31
"wday"	星期中第几天的数字表示	0（表示星期天）到 6（表示星期六）
"mon"	月份的数字表示	1 到 12
"year"	4 位数字表示的完整年份	例如：1999 或 2003
"yday"	一年中第几天的数字表示	0 到 365
"weekday"	星期几的完整文本表示	Sunday 到 Saturday
"month"	月份的完整文本表示	January 到 December
0	自从 Unix 纪元开始至今的秒数，和 time() 的返回值以及用于 date() 的值类似。	系统相关，典型值为从 -2147483648 到 2147483647。

例子

例子 1

```
<?php
print_r(getdate());
?>
```

输出：

```
Array
(
[seconds] => 45
[minutes] => 52
[hours] => 14
[mday] => 24
[wday] => 2
[mon] => 1
[year] => 2006
[yday] => 23
[weekday] => Tuesday
[month] => January
[0] => 1138110765
)
```

例子 2

```
<?php
$my_t=getdate(date("U"));
print("$my_t[weekday], $my_t[month] $my_t[mday], $my_t[year]");
?>
```

输出：

```
Wednesday, January 25, 2006
```

PHP gettimeofday() 函数

定义和用法

gettimeofday() 函数返回一个包含当前时间信息的数组。

自 PHP 5.1.0 起有个可选参数 *return_float*，当其设置为 TRUE 时，gettimeofday() 会返回一个浮点数。

所返回的数组键的含义是：

- "sec" - 自 Unix 纪元起的秒数
- "usec" - 微秒数
- "minuteswest" - 格林威治向西的分钟数
- "dsttime" - 夏令时修正的类型

语法

```
gettimeofday(return_float)
```

参数	描述
return_float	可选。当其设置为 TRUE 时，gettimeofday() 会返回一个浮点数。

例子

例子 1

```
<?php
echo(gettimeofday(true) . "<br /><br />");
print_r(gettimeofday());
?>
```

输出：

```
1138111447.4

Array
(
[sec] => 1138111447
[usec] => 395863
[minuteswest] => -60
[dsttime] => 0
)
```

例子 2

```
<?php
$my_t=gettimeofday();
print("$my_t[sec].$my_t[usec]");
?>
```

输出：

```
1138197006.988273
```

PHP gmdate() 函数

定义和用法

gmdate() 函数格式化 GMT/UTC 日期/时间。

同 date() 函数 类似，不同的是返回的时间是格林威治标准时（GMT）。

语法

```
gmdate(format,timestamp)
```

参数	描述
format	可选。规定如何返回结果。
timestamp	可选。

提示和注释

注释：在 PHP 5.1.0 之前，负的时间戳（1970 年之前的日期）在某些系统下（例如 Windows）不能工作。

例子

例子 1

当在中国（GMT +0800）运行以下程序时，第一行显示“Jan 01 2000 00:00:00”，而第二行显示“Dec 31 1999 16:00:00”。

```
<?php
echo date("M d Y H:i:s", mktime (0,0,0,1,1,2000));
echo gmdate("M d Y H:i:s", mktime (0,0,0,1,1,2000));
?>
```

输出：

```
Jan 01 2000 00:00:00
Dec 31 1999 16:00:00
```

例子 2

```
<?php
echo("Result with date():<br />");
echo(date("l") . "<br />");
echo(date("l dS \of F Y h:i:s A") . "<br />");
echo("Oct 3,1975 was on a ".date("l", mktime(0,0,0,10,3,1975))."<br />");
echo(date(DATE_RFC822) . "<br />");
echo(date(DATE_ATOM,mktime(0,0,0,10,3,1975)) . "<br /><br />");

echo("Result with gmdate():<br />");
echo(gmdate("l") . "<br />");
echo(gmdate("l dS \of F Y h:i:s A") . "<br />");
echo("Oct 3,1975 was on a ".gmdate("l", mktime(0,0,0,10,3,1975))."<br />");
echo(gmdate(DATE_RFC822) . "<br />");
echo(gmdate(DATE_ATOM,mktime(0,0,0,10,3,1975)) . "<br />");
?>
```

输出：

```
Result with date():
Tuesday
Tuesday 24th of January 2006 02:41:22 PM
Oct 3,1975 was on a Friday
Tue, 24 Jan 2006 14:41:22 CET
1975-10-03T00:00:00+0100

Result with gmdate():
Tuesday
Tuesday 24th of January 2006 01:41:22 PM
Oct 3,1975 was on a Thursday
Tue, 24 Jan 2006 13:41:22 GMT
1975-10-02T23:00:00+0000
```

PHP gmmktime() 函数

定义和用法

gmmktime() 函数取得 GMT 日期的 UNIX 时间戳。

与 mktime() 类似，不同的是返回值是格林威治标准时的时间戳。

参数总是表示 GMT 日期，因此 *is_dst* 对结果没有影响。

与 mktime() 一样，参数可以从右到左依次空着，空着的参数会被设为相应的当前 GMT 值。

语法

```
gmmktime(hour,minute,second,month,day,year,is_dst)
```

参数	描述
hour	可选。规定小时。
minute	可选。规定分钟。
second	可选。规定秒。
month	可选。规定用数字表示的月。
day	可选。规定天。
year	可选。规定年。在某些系统上，合法值介于 1901 - 2038 之间。不过在 PHP 5 中已经不存在这个限制了。
is_dst	可选。如果时间在日光节约时间(DST)期间，则设置为1，否则设置为0，若未知，则设置为-1。自 5.1.0 起，is_dst 参数被废弃。因此应该使用新的时区处理特性。

提示和注释

注释：gmmktime() 内部使用了 mktime()，因此只有转换成本地时间也合法的时间才能用在其中。

例子

```
<?php
$my_birthday = gmmktime(0,0,0,10,3,1975);
print($my_birthday . "<br />");
print(date("M-d-Y",$my_birthday));
?>
```

输出：

```
181526400
Oct-03-1975
```

PHP gmstrftime() 函数

定义和用法

gmstrftime() 函数根据本地区域设置格式化 GMT/UTC 时间／日期。

语法

```
gmstrftime(format,timestamp)
```

参数	描述
format	可选。规定如何返回结果。
timestamp	可选。

提示和注释

提示：与 strftime() 的行为相同，不同的是返回时间是格林威治标准时（GMT）。

例子

输出 strftime() 和 gmstrftime() 的结果：

```
<?php
echo(strftime("%b %d %Y %X", mktime(20,0,0,12,31,98)));
echo(gmstrftime("%b %d %Y %X", mktime(20,0,0,12,31,98)));

//输出当前日期、时间和时区
echo(gmstrftime("It is %a on %b %d, %Y, %X time zone: %Z",time()));
?>
```

输出：

```
Dec 31 1998 20:00:00
Dec 31 1998 19:00:00
It is Wed on Jan 25, 2006, 11:32:10 time zone: W. Europe Standard Time
```

PHP idate() 函数

定义和用法

idate() 函数将本地时间/日期格式化为整数。

语法

```
strftime(format,timestamp)
```

参数	描述
format	可选。规定如何返回结果。
timestamp	可选。

说明

根据给定的格式字符对 *timestamp* 进行格式化，并返回数字结果。

timestamp 为可选项，默认值为本地当前时间，即 time() 的值。

提示和注释

注释：与 date() 不同，idate() 只接受一个字符作为 *format* 参数。

format 参数可识别以下字符

format 字符	描述
B	Swatch Beat/Internet Time
d	月份中的第几天
h	小时（12 小时格式）
H	小时（24 小时格式）
i	分钟
I	如果启用夏时制则返回 1，否则返回 0
L	如果是闰年则返回 1，否则返回 0
m	月份的数字
s	秒数
t	本月的总天数
U	自 Unix 纪元（January 1 1970 00:00:00 GMT）起的秒数——这和 time() 作用相同
w	星期中的第几天（星期天是 0）
W	ISO-8601 格式年份中的第几个星期，每星期从星期一开始
y	年份（1 或 2 位数字——见下面说明）
Y	年份（4 位数字）
z	年份中的第几天
Z	以秒为单位的时区偏移量

例子

```
<?php
echo(idate("Y"));
?>
```

输出：

```
2008
```

PHP localtime() 函数

定义和用法

localtime() 函数返回本地时间（一个数组）。

localtime() 的第一个参数是时间戳，如果没有给出则使用从 time() 返回的当前时间。

第二个参数是 *is_associative*，如果设为 false 或未提供则返回的是普通的数字索引数组。如果该参数设为 true 则 localtime() 函数返回一个关联数组。

关联数组中不同的键名是：

- "tm_sec" - 秒数
- "tm_min" - 分钟数
- "tm_hour" - 小时
- "tm_mday" - 月份中的第几日
- "tm_mon" - 年份中的第几个月，从 0 开始表示一月
- "tm_year" - 年份，从 1900 开始
- "tm_wday" - 星期中的第几天
- "tm_yday" - 一年中的第几天
- "tm_isdst" - 夏令时当前是否生效

注释：月份从 0（一月）到 11（十二月），星期数从 0（星期天）到 6（星期六）。

语法

```
localtime(timestamp,is_associative)
```

参数	描述
timestamp	可选。规定被格式化的日期或时间。若未规定 timestamp，则使用当前的本地时间。
is_associative	可选。规定返回索引数组还是关联数组。

例子

```
<?php
$localtime = localtime();
$localtime_assoc = localtime(time(), true);
print_r($localtime);
print_r($localtime_assoc);
?>
```

输出：

```
Array
(
    [0] => 24
    [1] => 3
    [2] => 19
    [3] => 3
    [4] => 3
    [5] => 105
    [6] => 0
    [7] => 92
    [9] => 1
)

Array
(
    [tm_sec] => 24
    [tm_min] => 3
    [tm_hour] => 19
    [tm_mday] => 3
    [tm_mon] => 3
    [tm_year] => 105
    [tm_wday] => 0
    [tm_yday] => 92
    [tm_isdst] => 1
)
```

PHP microtime() 函数

定义和用法

microtime() 函数返回当前 Unix 时间戳和微秒数。

语法

```
microtime(get_as_float)
```

参数	描述
get_as_float	如果给出了 get_as_float 参数并且其值等价于 TRUE，该函数将返回一个浮点数。

说明

本函数仅在支持 gettimeofday() 系统调用的操作系统下可用。

如果调用时不带可选参数，本函数以 "msec sec" 的格式返回一个字符串，其中 sec 是自 Unix 纪元（0:00:00 January 1, 1970 GMT）起到现在的秒数，msec 是微秒部分。字符串的两部分都是以秒为单位返回的。

例子

```
<?php
echo(microtime());
?>
```

输出：

```
0.25139300 1138197510
```

PHP mktime() 函数

定义和用法

mktime() 函数返回一个日期的 Unix 时间戳。

参数总是表示 GMT 日期，因此 *is_dst* 对结果没有影响。

参数可以从右到左依次空着，空着的参数会被设为相应的当前 GMT 值。

语法

```
mktime(hour,minute,second,month,day,year,is_dst)
```

参数	描述
hour	可选。规定小时。
minute	可选。规定分钟。
second	可选。规定秒。
month	可选。规定用数字表示的月。
day	可选。规定天。
year	可选。规定年。在某些系统上，合法值介于 1901 - 2038 之间。不过在 PHP 5 中已经不存在这个限制了。
is_dst	可选。如果时间在日光节约时间(DST)期间，则设置为1，否则设置为0，若未知，则设置为-1。自 5.1.0 起，is_dst 参数被废弃。因此应该使用新的时区处理特性。

提示和注释

注释：在 PHP 5.1 之前，如果该函数的参数非法，则会返回 false。

例子

mktime() 函数对于日期运算和验证非常有用。它可以自动校正越界的输入：

```
<?php
echo(date("M-d-Y",mktime(0,0,0,12,36,2001)));
echo(date("M-d-Y",mktime(0,0,0,14,1,2001)));
echo(date("M-d-Y",mktime(0,0,0,1,1,2001)));
echo(date("M-d-Y",mktime(0,0,0,1,1,99)));
?>
```

输出：

```
Jan-05-2002
Feb-01-2002
Jan-01-2001
Jan-01-1999
```

PHP strftime() 函数

定义和用法

strftime() 函数根据区域设置格式化本地时间／日期。

语法

```
strftime(format,timestamp)
```

参数	描述
format	可选。规定如何返回结果。
timestamp	可选。

提示和注释

提示：与 gmstrftime() 的行为相同，不同的是返回时间是本地时间。

例子

输出 strftime() 和 gmstrftime() 的结果：

```
<?php
echo(strftime("%b %d %Y %X", mktime(20,0,0,12,31,98)));
echo(gmstrftime("%b %d %Y %X", mktime(20,0,0,12,31,98)));

//输出当前日期、时间和时区
echo(gmstrftime("It is %a on %b %d, %Y, %X time zone: %Z",time()));
?>
```

输出：

```
Dec 31 1998 20:00:00
Dec 31 1998 19:00:00
It is Wed on Jan 25, 2006, 11:32:10 time zone: W. Europe Standard Time
```

PHP strptime() 函数

定义和用法

strptime() 函数解析由 strftime() 生成的日期／时间。

语法

```
strptime(date,format)
```

参数	描述
date	要解析的字符串（例如从 strftime() 返回的）。
format	date 所使用的格式（与 strftime() 中所使用的相同）。

说明

strptime() 返回一个将 *date* 解析后的数组，如果出错返回 FALSE。

月份和星期几的名字以及其它与语种有关的字符串对应于 setlocale()设定的当前区域（LC_TIME）。

数组中包含以下单元：

键名	说明
tm_sec	当前分钟内的秒数（0-61）
tm_min	当前小时内的分钟数（0-59）
tm_hour	午夜起的小时数（0-23）
tm_mday	月份中的第几天（1-31）
tm_mon	自一月起过了几个月（0-11）
tm_year	自 1900 年起过了几年
tm_wday	自星期天起过了几天（0-6）
tm_yday	本年自一月一日起过了多少天（0-365）
unparsed date	中未能通过指定的 format 识别的部分

例子

输出 strftime() 和 strptime() 的结果：

```
<?php
$format="%d/%m/%Y %H:%M:%S";
$strf=strftime($format);
echo("$strf");
print_r(strptime($strf,$format));
?>
```

输出：

```
03/10/2005 13:23:44
Array
(
[tm_sec] => 44
[tm_min] => 23
[tm_hour] => 13
[tm_mday] => 3
[tm_mon] => 9
[tm_year] => 105
[tm_wday] => 0
[tm_yday] => 276
[unparsed] =>
)
```

PHP strtotime() 函数

定义和用法

strtotime() 函数将任何英文文本的日期时间描述解析为 Unix 时间戳。

语法

```
strtotime(time,now)
```

参数	描述
time	规定要解析的时间字符串。
now	用来计算返回值的时间戳。如果省略该参数，则使用当前时间。

说明

该函数预期接受一个包含美国英语日期格式的字符串并尝试将其解析为 Unix 时间戳（自 January 1 1970 00:00:00 GMT 起的秒数），其值相对于 *now* 参数给出的时间，如果没有提供此参数，则用系统当前时间。

该函数将使用 *TZ* 环境变量（如果有的话）来计算时间戳。自 PHP 5.1.0 起有更容易的方法来定义时区用于所有的日期／时间函数。此过程在 date_default_timezone_get() 函数页面中有说明。

返回值

成功则返回时间戳，否则返回 FALSE。在 PHP 5.1.0 之前本函数在失败时返回 -1。

例子

```
<?php
echo(strtotime("now"));
echo(strtotime("3 October 2005"));
echo(strtotime("+5 hours"));
echo(strtotime("+1 week"));
echo(strtotime("+1 week 3 days 7 hours 5 seconds"));
echo(strtotime("next Monday"));
echo(strtotime("last Sunday"));
?>
```

输出：

```
1138614504
1128290400
1138632504
1139219304
1139503709
1139180400
1138489200
```

PHP time() 函数

定义和用法

time() 函数返回当前时间的 Unix 时间戳。

语法

```
time(void)
```

参数	描述
void	可选。

说明

返回自从 Unix 纪元（格林威治时间 1970 年 1 月 1 日 00:00:00）到当前时间的秒数。

提示和注释

提示：自 PHP 5.1 起在 *$_SERVER['REQUEST_TIME']* 中保存了发起该请求时刻的时间戳。

例子

例子 1

```
<?php
$t=time();
echo($t . "<br />");
echo(date("D F d Y",$t));
?>
```

输出：

```
1138618081
Mon January 30 2006
```

例子 2

```
<?php
$nextWeek = time() + (7 * 24 * 60 * 60); // 7 days; 24 hours; 60 mins; 60secs
echo 'Now:       '. date('Y-m-d') ."\n";
echo 'Next Week: '. date('Y-m-d', $nextWeek) ."\n";
?>
```

输出：

```
Now:       2005-03-30
Next Week: 2005-04-07
```

PHP Directory 函数

PHP Directory 简介

Directory 函数允许您获得关于目录及其内容的信息。

安装

Directory 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP Directory 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
chdir()	改变当前的目录。	3
chroot()	改变当前进程的根目录。	4
dir()	打开一个目录句柄，并返回一个对象。	3
closedir()	关闭目录句柄。	3
getcwd()	返回当前目录。	4
opendir()	打开目录句柄。	3
readdir()	返回目录句柄中的条目。	3
rewinddir()	重置目录句柄。	3
scandir()	列出指定路径中的文件和目录。	5

PHP Directory 常量

PHP：指示支持常量的最早的 PHP 版本。

常量	描述	PHP
DIRECTORY_SEPARATOR	3	
PATH_SEPARATOR	4	

PHP chdir() 函数

定义和用法

chdir() 函数把当前的目录改变为指定的目录。

若成功，则该函数返回 true，否则返回 false。

语法

```
chdir(directory)
```

参数	描述
directory	必需。规定新的当前目录。

例子

```
<?php
//获得当前目录
echo getcwd();
echo "<br />";

//改变为 images 目录
chdir("images");
echo "<br />";
echo getcwd();
?>
```

输出：

```
C:\testweb\main
C:\testweb\main\images
```

PHP chroot() 函数

定义和用法

chroot() 函数把当前进程的根目录改变为指定的目录。

若成功，则该函数返回 true，否则返回 false。

语法

```
chroot(directory)
```

参数	描述
directory	必需。规定新的根目录。

提示和注释

提示：该函数没有在 Windows 平台上实现。

PHP dir() 函数

定义和用法

dir() 函数打开一个目录句柄，并返回一个对象。这个对象包含三个方法：read() , rewind() 以及 close()。

若成功，则该函数返回一个目录流，否则返回 false 以及一个 error。可以通过在函数名前加上 "@" 来隐藏 error 的输出。

语法

```
dir(directory)
```

参数	描述
directory	必需。规定目录。

例子

例子 1

```
<?php
//打开 images 目录
$dir = dir("images");

//列出 images 目录中的文件
while (($file = $dir->read()) !== false)
{
echo "filename: " . $file . "<br />";
}

$dir->close();
?>
```

输出：

```
filename: .
filename: ..
filename: cat.gif
filename: dog.gif
filename: food
filename: horse.gif
```

例子 2

如果 dir() 的输出失败，本例会隐藏错误：

```
<?php
//打开 images 目录
$dir = @ dir("images");

//列出 images 目录中的文件
while (($file = $dir->read()) !== false)
  {
  echo "filename: " . $file . "<br />";
  }

$dir->close();
?>
```

输出：

```
filename: .
filename: ..
filename: cat.gif
filename: dog.gif
filename: food
filename: horse.gif
```

PHP closedir() 函数

定义和用法

closedir() 函数关闭由 opendir() 函数打开的目录句柄。

语法

```
closedir(dir_stream)
```

参数	描述
dir_stream	必需。规定要关闭的目录句柄。

例子

```
<?php
//打开 images 目录
$dir = opendir("images");

//列出 images 目录中的文件
while (($file = readdir($dir)) !== false)
  {
  echo "filename: " . $file . "<br />";
  }
  closedir($dir);
?>
```

输出：

```
filename: .
filename: ..
filename: cat.gif
filename: dog.gif
filename: food
filename: horse.gif
```

PHP getcwd() 函数

定义和用法

getcwd() 函数返回当前目录。

若成功，则返回当前工作目录，否则返回 false。

语法

```
getcwd()
```

例子

```
<?php
echo getcwd();
?>
```

输出：

```
C:\testweb\main
```

PHP opendir() 函数

定义和用法

opendir() 函数打开一个目录句柄，可由 closedir()，readdir() 和 rewinddir() 使用。

若成功，则该函数返回一个目录流，否则返回 false 以及一个 error。可以通过在函数名前加上 "@" 来隐藏 error 的输出。

语法

```
opendir(path,context)
```

参数	描述
path	必需。规定要打开的目录路径。
context	可选。规定目录句柄的环境。context 是可修改目录流的行为的一套选项。

提示和注释

注释：从 PHP 5.0.0 开始，*path* 参数支持 ftp:// URL wrapper。

注释：在 PHP 4.3.0 中， *path* 参数可以是任何支持目录列表的 URL，不过在 PHP 4 中只有 file:// URL wrapper 支持此功能。

例子

例子 1

```
<?php
//打开 images 目录
$dir = opendir("images");

//列出 images 目录中的文件
while (($file = readdir($dir)) !== false)
  {
  echo "filename: " . $file . "<br />";
  }
  closedir($dir);
?>
```

输出：

```
filename: .
filename: ..
filename: cat.gif
filename: dog.gif
filename: food
filename: horse.gif
```

例子 2

如果 opendir() 的输出失败，本例会隐藏错误：

```
<?php
//打开 images 目录
$dir = @ opendir("images");

//列出 images 目录中的文件
while (($file = readdir($dir)) !== false)
  {
  echo "filename: " . $file . "<br />";
  }
  closedir($dir);
?>
```

输出：

```
filename: .
filename: ..
filename: cat.gif
filename: dog.gif
filename: food
filename: horse.gif
```

PHP readdir() 函数

定义和用法

readdir() 函数返回由 opendir() 打开的目录句柄中的条目。

若成功，则该函数返回一个文件名，否则返回 false。

语法

```
readdir(dir_stream)
```

参数	描述
dir_stream	必需。规定要使用的目录句柄。

说明

返回目录中下一个文件的文件名。文件名以在文件系统中的排序返回。

例子

```
<?php
//打开 images 目录
$dir = opendir("images");

//列出 images 目录中的文件
while (($file = readdir($dir)) !== false)
  {
  echo "filename: " . $file . "<br />";
  }
  closedir($dir);
?>
```

输出：

```
filename: .
filename: ..
filename: cat.gif
filename: dog.gif
filename: food
filename: horse.gif
```

PHP rewinddir() 函数

定义和用法

rewinddir() 函数重置由 opendir() 打开的目录句柄。

本函数什么都不会返回。

语法

```
rewinddir(dir_stream)
```

参数	描述
dir_stream	必需。规定要使用的目录句柄。

提示和注释

提示：本函数重读目录，并可用于检查目录中的变化。

例子

```
<?php
//打开 images 目录
$dir = opendir("images");

//列出 images 目录中的文件
while (($file = readdir($dir)) !== false)
  {
  echo "filename: " . $file . "<br />";
  }

//重置目录流
rewinddir($dir);

//供检查变化的代码

closedir($dir);
?>
```

PHP scandir() 函数

定义和用法

scandir() 函数返回一个数组，其中包含指定路径中的文件和目录。

若成功，则返回一个数组，若失败，则返回 false。如果 *directory* 不是目录，则返回布尔值 false 并生成一条 E_WARNING 级的错误。

语法

```
scandir(directory,sort,context)
```

参数	描述
directory	必需。规定要扫描的目录。
sort	可选。规定排列顺序。默认是 0 （升序）。如果是 1，则为降序。
context	可选。规定目录句柄的环境。context 是可修改目录流的行为的一套选项。

例子

```
<?php
print_r(scandir("images"));
?>
```

输出：

```
Array
(
[0] => .
[1] => ..
[2] => dog.jpg
[3] => house.jpg
[4] => logo.gif
)
```

PHP Error 和 Logging 函数

PHP Error 和 Logging 简介

error 和 logging 函数允许你对错误进行处理和记录。

error 函数允许用户定义错误处理规则，并修改记录错误的方式。

logging 函数允许用户对应用程序进行日志记录，并把日志消息发送到电子邮件、系统日志或其他的机器。

安装

error 和 logging 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP Error 和 Logging 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
debug_backtrace()	生成 backtrace。	4
debug_print_backtrace()	输出 backtrace。	5
error_get_last()	获得最后发生的错误。	5
error_log()	向服务器错误记录、文件或远程目标发送一个错误。	4
error_reporting()	规定报告哪个错误。	4
restore_error_handler()	恢复之前的错误处理程序。	4
restore_exception_handler()	恢复之前的异常处理程序。	5
set_error_handler()	设置用户自定义的错误处理函数。	4
set_exception_handler()	设置用户自定义的异常处理函数。	5
trigger_error()	创建用户自定义的错误消息。	4
user_error()	trigger_error() 的别名。	4

PHP Error 和 Logging 常量

PHP：指示支持该常量的最早的 PHP 版本。

值	常量	描述	PHP
1	E_ERROR	致命的运行时错误。错误无法恢复。脚本的执行被中断。	
2	E_WARNING	非致命的运行时错误。脚本的执行不会中断。	
4	E_PARSE	编译时语法解析错误。解析错误只应该由解析器生成。	
8	E_NOTICE	运行时提示。可能是错误，也可能在正常运行脚本时发生。	
16	E_CORE_ERROR	由 PHP 内部生成的错误。	4
32	E_CORE_WARNING	由 PHP 内部生成的警告。	4
64	E_COMPILE_ERROR	由 Zend 脚本引擎内部生成的错误。	4
128	E_COMPILE_WARNING	由 Zend 脚本引擎内部生成的警告。	4
256	E_USER_ERROR	由于调用 trigger_error() 函数生成的运行时错误。	4
512	E_USER_WARNING	由于调用 trigger_error() 函数生成的运行时警告。	4
1024	E_USER_NOTICE	由于调用 trigger_error() 函数生成的运行时提示。	4
2048	E_STRICT	运行时提示。对增强代码的互用性和兼容性有益。	5
4096	E_RECOVERABLE_ERROR	可捕获的致命错误。（参阅 set_error_handler()）	5
8191	E_ALL	所有的错误和警告，除了 E_STRICT。	5

PHP debug_backtrace() 函数

定义和用法

PHP debug_backtrace() 函数生成一个 backtrace。

该函数返回一个关联数组。下面是可能返回的元素：

名称	类型	描述
function	字符串	当前的函数名。
line	整数	当前的行号。
file	字符串	当前的文件名。
class	字符串	当前的类名
object	对象	当前对象。
type	字符串	当前的调用类型，可能的调用： 返回: " `->` " - 方法调用 返回: " `::` " - 静态方法调用 返回 `nothing` - 函数调用
args	数组	如果在函数中，列出函数参数。如果在被引用的文件中，列出被引用的文件名。

语法

```
debug_backtrace()
```

例子

```
<?php
function one($str1, $str2)
 {
 two("Glenn", "Quagmire");
 }
function two($str1, $str2)
 {
 three("Cleveland", "Brown");
 }
function three($str1, $str2)
 {
 print_r(debug_backtrace());
 }

one("Peter", "Griffin");
?>
```

输出：

```
Array
(
[0] => Array
 (
 [file] => C:\webfolder\test.php
 [line] => 7
 [function] => three
 [args] => Array
  (
  [0] => Cleveland
  [1] => Brown
  )
 )
[1] => Array
 (
 [file] => C:\webfolder\test.php
 [line] => 3
 [function] => two
 [args] => Array
  (
  [0] => Glenn
  [1] => Quagmire
  )
 )
[2] => Array
 (
 [file] => C:\webfolder\test.php
 [line] => 14
 [function] => one
 [args] => Array
  (
  [0] => Peter
  [1] => Griffin
  )
 )
)
```

PHP debug_print_backtrace() 函数

定义和用法

debug_print_backtrace() 函数输出 backtrace。

语法

```
debug_print_backtrace()
```

例子

```
<?php
function one($str1, $str2)
 {
 two("Glenn", "Quagmire");
 }
function two($str1, $str2)
 {
 three("Cleveland", "Brown");
 }
function three($str1, $str2)
 {
 debug_print_backtrace();
 }

one("Peter", "Griffin");
?>
```

输出：

```
#0 three(Cleveland, Brown) called at [C:\webfolder\test.php:8]
#1 two(Glenn, Quagmire) called at [C:\webfolder\test.php:4]
#2 one(Peter, Griffin) called at [C:\webfolder\test.php:15]
```

PHP error_get_last() 函数

定义和用法

error_get_last() 函数获取最后发生的错误。

该函数以数组的形式返回最后发生的错误。

返回的数组包含 4 个键和值：

- [type] - 错误类型
- [message] - 错误消息
- [file] - 发生错误所在的文件
- [line] - 发生错误所在的行

语法

```
error_get_last()
```

例子

```
<?php
echo $test;
print_r(error_get_last());
?>
```

输出：

```
Array
(
[type] => 8
[message] => Undefined variable: test
[file] => C:\webfolder\test.php
[line] => 2
)
```

PHP error_log() 函数

定义和用法

error_log() 函数向服务器错误记录、文件或远程目标发送一个错误。

若成功，返回 true，否则返回 false。

语法

```
error_log(error,type,destination,headers)
```

参数	描述
error	必需。要记录的错误消息。
type	可选。规定错误记录的类型。可能的记录类型： `0` - 默认。根据在 php.ini 文件中的 error_log 配置，错误被发送到服务器日志系统或文件。 `1` - 错误被发送到 destination 参数中的地址。只有该类型使用 headers 参数。 `2` - 通过 PHP debugging 连接来发送错误。该选项只在 PHP 3 中可用。 `3` - 错误发送到文件目标字符串。
destination	可选。规定向何处发送错误消息。该参数的值依赖于 "type" 参数的值。
headers	可选。只在 "type" 为 1 时使用。规定附加的头部，比如 From, Cc 以及 Bcc。由 CRLF (\r\n) 分隔。注释：在发送电子邮件时，必须包含 From 头部。可以在 php.ini 文件中或者通过此参数设置。

例子

本例发送一封带有自定义错误的电子邮件：

```
<?php
$test=2;

if ($test>1)
{
error_log("A custom error has been triggered",
1,"someone@example.com","From: webmaster@example.com");
}
?>
```

输出：

```
A custom error has been triggered
```

PHP error_reporting() 函数

定义和用法

error_reporting() 设置 PHP 的报错级别并返回当前级别。

语法

```
error_reporting(report_level)
```

如果参数 level 未指定，当前报错级别将被返回。下面几项是 level 可能的值：

值	常量	描述
1	E_ERROR	Fatal run-time errors. Errors that can not be recovered from. Execution of the script is halted
2	E_WARNING	Non-fatal run-time errors. Execution of the script is not halted
4	E_PARSE	Compile-time parse errors. Parse errors should only be generated by the parser
8	E_NOTICE	Run-time notices. The script found something that might be an error, but could also happen when running a script normally
16	E_CORE_ERROR	Fatal errors at PHP startup. This is like an E_ERROR in the PHP core
32	E_CORE_WARNING	Non-fatal errors at PHP startup. This is like an E_WARNING in the PHP core
64	E_COMPILE_ERROR	Fatal compile-time errors. This is like an E_ERROR generated by the Zend Scripting Engine
128	E_COMPILE_WARNING	Non-fatal compile-time errors. This is like an E_WARNING generated by the Zend Scripting Engine
256	E_USER_ERROR	Fatal user-generated error. This is like an E_ERROR set by the programmer using the PHP function trigger_error()
512	E_USER_WARNING	Non-fatal user-generated warning. This is like an E_WARNING set by the programmer using the PHP function trigger_error()
1024	E_USER_NOTICE	User-generated notice. This is like an E_NOTICE set by the programmer using the PHP function trigger_error()
2048	E_STRICT	Run-time notices. PHP suggest changes to your code to help interoperability and compatibility of the code
4096	E_RECOVERABLE_ERROR	Catchable fatal error. This is like an E_ERROR but can be caught by a user defined handle (see also set_error_handler())
8191	E_ALL	All errors and warnings, except level E_STRICT (E_STRICT will be part of E_ALL as of PHP 6.0)

例子

任意数目的以上选项都可以用“或”来连接（用 OR 或 |），这样可以报告所有需要的各级别错误。例如，下面的代码关闭了用户自定义的错误和警告，执行了某些操作，然后恢复到原始的报错级别：

```
<?php
//禁用错误报告
error_reporting(0);

//报告运行时错误
error_reporting(E_ERROR | E_WARNING | E_PARSE);

//报告所有错误
error_reporting(E_ALL);
?>
```

PHP restore_error_handler() 函数

定义和用法

restore_error_handler() 函数恢复之前的错误处理程序，该程序是由 set_error_handler() 函数改变的。

该函数永远返回 true。

语法

```
restore_error_handler()
```

提示和注释

提示：之前的错误处理程序可能是在错误处理程序或用户自定义函数中构建的。

例子

```
<?php
//custom error handler function
function customError($errno, $errstr, $errfile, $errline)
 {
 echo "<b>Custom error:</b> [$errno] $errstr<br />";
 echo " Error on line $errline in $errfile<br />";
 }

//set user-defined error handler
set_error_handler("customError");

$test=2;

//trigger error
if ($test>1)
 {
 trigger_error("A custom error has been triggered");
 }

//restore built-in error handler
restore_error_handler

//trigger error again
if ($test>1)
 {
 trigger_error("A custom error has been triggered");
 }
?>
```

输出：

```
**Custom error:** [1024] A custom error has been triggered
Error on line 14 in C:\webfolder\test.php

**Notice**: A custom error has been triggered in **
C:\webfolder\test.php** on line **21**
```

PHP restore_exception_handler() 函数

定义和用法

restore_exception_handler() 函数恢复之前的异常处理程序，该程序是由 set_exception_handler() 函数改变的。

该函数永远返回 true。

语法

```
restore_exception_handler()
```

提示和注释

提示：之前的异常处理程序可能是在异常处理程序或用户自定义函数中构建的。

例子

```
<?php
restore_exception_handler();

throw new Exception('Uncaught Exception occured');
?>
```

输出：

```
**Fatal error**: Uncaught exception 'Exception' with message
'Uncaught Exception occured' in C:\webfolder\test.php:4
Stack trace: #0 {main} thrown in **C:\webfolder\test.php** on line **4**
```

PHP set_error_handler() 函数

定义和用法

set_error_handler() 函数设置用户自定义的错误处理函数。

该函数用于创建运行时期间的用户自己的错误处理方法。

该函数会返回旧的错误处理程序，若失败，则返回 null。

语法

```
set_error_handler(error_function,error_types)
```

参数	描述
error_function	必需。规定发生错误时运行的函数。
error_types	可选。规定在哪个错误报告级别会显示用户定义的错误。默认是 "E_ALL"。

提示和注释

提示：如果使用了该函数，会完全绕过标准的 PHP 错误处理函数，如果必要，用户定义的错误处理程序必须终止 (die()) 脚本。

注释：如果在脚本执行前发生错误，由于在那时自定义程序还没有注册，因此就不会用到这个自定义错误处理程序。

例子

```
<?php
//error handler function
function customError($errno, $errstr, $errfile, $errline)
 {
 echo "<b>Custom error:</b> [$errno] $errstr<br />";
 echo " Error on line $errline in $errfile<br />";
 echo "Ending Script";
 die();
 }

//set error handler
set_error_handler("customError");

$test=2;

//trigger error
if ($test>1)
 {
 trigger_error("A custom error has been triggered");
 }
?>
```

输出：

```
**Custom error:** [1024] A custom error has been triggered
Error on line 19 in C:\webfolder\test.php
Ending Script
```

PHP set_exception_handler() 函数

定义和用法

set_exception_handler() 函数设置用户自定义的异常处理函数。

该函数用于创建运行时期间的用户自己的异常处理方法。

该函数会返回旧的异常处理程序，若失败，则返回 null。

语法

```
set_exception_handler(exception_function)
```

参数	描述
error_function	必需。规定未捕获的异常发生时调用的函数。该函数必须在调用 set_exception_handler() 函数之前定义。这个异常处理函数需要需要一个参数，即抛出的 exception 对象。

提示和注释

提示：在这个异常处理程序被调用后，脚本会停止执行。

例子

```
<?php
function myException($exception)
{
echo "<b>Exception:</b> " , $exception->getMessage();
}

set_exception_handler('myException');

throw new Exception('Uncaught Exception occurred');
?>
```

输出：

```
**Exception:** Uncaught Exception occurred
```

PHP trigger_error() 函数

定义和用法

trigger_error() 函数创建用户定义的错误消息。

trigger_error() 用于在用户指定的条件下触发一个错误消息。它与内建的错误处理器一同使用，也可以与由 set_error_handler() 函数创建的用户自定义函数使用。

如果指定了一个不合法的错误类型，该函数返回 false，否则返回 true。

语法

```
trigger_error(error_message,error_types)
```

参数	描述
error_message	必需。规定错误消息。长度限制为 1024 个字符。
error_types	可选。规定错误消息的错误类型。 可能的值： `E_USER_ERROR` `E_USER_WARNING` `E_USER_NOTICE`

例子

```
<?php
$test=2;
if ($test>1)
{
trigger_error("A custom error has been triggered");
}
?>
```

输出：

```
**Notice**: A custom error has been triggered
in **C:\webfolder\test.php** on line **6**
```

PHP Filesystem 函数

PHP Filesystem 简介

Filesystem 函数允许您访问和操作文件系统。

安装

Filesystem 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

Runtime 配置

文件系统函数的行为受到 php.ini 中设置的影响。

文件系统配置选项：

名称	默认	描述	可改变
allow_url_fopen	"1"	本选项激活了 URL 形式的 fopen 封装协议使得可以访问 URL 对象例如文件。默认的封装协议提供用 ftp 和 http 协议来访问远程文件，一些扩展库例如 zlib 可能会注册更多的封装协议。（PHP 4.0.4 版以后可用。）	PHP_INI_SYSTEM
user_agent	NULL	定义 PHP 发送的 User-Agent。（PHP 4.3.0 版以后可用。）	PHP_INI_ALL
default_socket_timeout	"60"	基于 socket 的流的默认超时时间(秒)。（PHP 4.3.0 版以后可用。）	PHP_INI_ALL
from	""	定义匿名 ftp 的密码（您的 email 地址）。	PHP_INI_ALL
auto_detect_line_endings	"0"	当设为 On 时，PHP 将检查通过 fgets() 和 file() 取得的数据中的行结束符号是符合 Unix，MS-DOS，还是 Macintosh 的习惯。这使得 PHP 可以和 Macintosh 系统交互操作，但是默认值是 Off，因为在检测第一行的 EOL 习惯时会有很小的性能损失，而且在 Unix 系统下使用回车符号作为项目分隔符的人们会遭遇向下不兼容的行为。（PHP 4.3.0 版以后可用。）	PHP_INI_ALL

Unix / Windows 兼容性

当在 Unix 平台上规定路径时，正斜杠 (/) 用作目录分隔符。而在 Windows 平台上，正斜杠 (/) 和反斜杠 () 均可使用。

PHP Filesystem 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
basename()	返回路径中的文件名部分	3

basename()	返回路径中的文件名部分。	3
chgrp()	改变文件组。	3
chmod()	改变文件模式。	3
chown()	改变文件所有者。	3
clearstatcache()	清除文件状态缓存。	3
copy()	复制文件。	3
delete()	参见 unlink() 或 unset()。	
dirname()	返回路径中的目录名称部分。	3
disk_free_space()	返回目录的可用空间。	4
disk_total_space()	返回一个目录的磁盘总容量。	4
diskfreespace()	disk_free_space() 的别名。	3
fclose()	关闭打开的文件。	3
feof()	测试文件指针是否到了文件结束的位置。	3
fflush()	向打开的文件输出缓冲内容。	4
fgetc()	从打开的文件中返回字符。	3
fgetcsv()	从打开的文件中解析一行，校验 CSV 字段。	3
fgets()	从打开的文件中返回一行。	3
fgetss()	从打开的文件中读取一行并过滤掉 HTML 和 PHP 标记。	3
file()	把文件读入一个数组中。	3
file_exists()	检查文件或目录是否存在。	3
file_get_contents()	将文件读入字符串。	4
file_put_contents()	将字符串写入文件。	5
fileatime()	返回文件的上次访问时间。	3
filectime()	返回文件的上次改变时间。	3
filegroup()	返回文件的组 ID。	3
fileinode()	返回文件的 inode 编号。	3
filemtime()	返回文件的上次修改时间。	3
fileowner()	文件的 user ID （所有者）。	3
fileperms()	返回文件的权限。	3
filesize()	返回文件大小。	3
filetype()	返回文件类型。	3

flock()	锁定或释放文件。	3
fnmatch()	根据指定的模式来匹配文件名或字符串。	4
fopen()	打开一个文件或 URL。	3
fpassthru()	从打开的文件中读数据，直到 EOF，并向输出缓冲写结果。	3
fputcsv()	将行格式化为 CSV 并写入一个打开的文件中。	5
fputs()	fwrite() 的别名。	3
fread()	读取打开的文件。	3
fscanf()	根据指定的格式对输入进行解析。	4
fseek()	在打开的文件中定位。	3
fstat()	返回关于一个打开的文件的信息。	4
ftell()	返回文件指针的读/写位置	3
ftruncate()	将文件截断到指定的长度。	4
fwrite()	写入文件。	3
glob()	返回一个包含匹配指定模式的文件名/目录的数组。	4
is_dir()	判断指定的文件名是否是一个目录。	3
is_executable()	判断文件是否可执行。	3
is_file()	判断指定文件是否为常规的文件。	3
is_link()	判断指定的文件是否是连接。	3
is_readable()	判断文件是否可读。	3
is_uploaded_file()	判断文件是否是通过 HTTP POST 上传的。	3
is_writable()	判断文件是否可写。	4
is_writeable()	is_writable() 的别名。	3
link()	创建一个硬连接。	3
linkinfo()	返回有关一个硬连接的信息。	3
lstat()	返回关于文件或符号连接的信息。	3
mkdir()	创建目录。	3
move_uploaded_file()	将上传的文件移动到新位置。	4
parse_ini_file()	解析一个配置文件。	4
pathinfo()	返回关于文件路径的信息。	4
pclose()	关闭有 popen() 打开的进程。	3
popen()	打开一个进程。	3

readfile()	读取一个文件，并输出到输出缓冲。	3
readlink()	返回符号连接的目标。	3
realpath()	返回绝对路径名。	4
rename()	重名名文件或目录。	3
rewind()	倒回文件指针的位置。	3
rmdir()	删除空的目录。	3
set_file_buffer()	设置已打开文件的缓冲大小。	3
stat()	返回关于文件的信息。	3
symlink()	创建符号连接。	3
tempnam()	创建唯一的临时文件。	3
tmpfile()	建立临时文件。	3
touch()	设置文件的访问和修改时间。	3
umask()	改变文件的文件权限。	3
unlink()	删除文件。	3

PHP Filesystem 常量

PHP：指示支持该常量的最早的 PHP 版本。

常量	描述	PHP
GLOB_BRACE		
GLOB_ONLYDIR		
GLOB_MARK		
GLOB_NOSORT		
GLOB_NOCHECK		
GLOB_NOESCAPE		
PATHINFO_DIRNAME		
PATHINFO_BASENAME		
PATHINFO_EXTENSION		
FILE_USE_INCLUDE_PATH		
FILE_APPEND		
FILE_IGNORE_NEW_LINES		
FILE_SKIP_EMPTY_LINES		

PHP basename() 函数

定义和用法

basename() 函数返回路径中的文件名部分。

语法

```
basename(path,suffix)
```

参数	描述
path	必需。规定要检查的路径。
suffix	可选。规定文件扩展名。如果文件有 suffix，则不会输出这个扩展名。

例子

```
<?php
$path = "/testweb/home.php";

//显示带有文件扩展名的文件名
echo basename($path);

//显示不带有文件扩展名的文件名
echo basename($path,".php");
?>
```

输出：

```
home.php
home
```

PHP chgrp() 函数

定义和用法

chgrp() 函数改变文件所属的组。

如果成功则返回 TRUE，否则返回 FALSE。

语法

```
chgrp(file,group)
```

参数	描述
file	必需。规定要检查的文件。
group	可选。规定新的组。可以是组名或组的 ID。

说明

尝试将文件 *file* 所属的组改成 *group* （通过组名或组 ID 指定）。

只有超级用户可以任意修改文件的组，其它用户可能只能将文件的组改成该用户自己所在的组。

例子

```
<?php
chgrp("test.txt","admin")
?>
```

PHP chmod() 函数

定义和用法

chmod() 函数改变文件模式。

如果成功则返回 TRUE，否则返回 FALSE。

语法

```
chmod(file,mode)
```

参数	描述
file	必需。规定要检查的文件。
mode	可选。规定新的权限。mode 参数由 4 个数字组成： 第一个数字永远是 0 第二个数字规定所有者的权限 第二个数字规定所有者所属的用户组的权限 第四个数字规定其他所有人的权限 可能的值（如需设置多个权限，请对下面的数字进行总计）： 1 - 执行权限 2 - 写权限 4 - 读权限

例子

```
<?php
// 所有者可读写，其他人没有任何权限
chmod("test.txt",0600);

// 所有者可读写，其他人可读
chmod("test.txt",0644);

// 所有者有所有权限，其他所有人可读和执行
chmod("test.txt",0755);

// 所有者有所有权限，所有者所在的组可读
chmod("test.txt",0740);
?>
```

PHP chown() 函数

定义和用法

chown() 函数改变指定文件的所有者。

如果成功则返回 TRUE，否则返回 FALSE。

语法

```
chown(file,owner)
```

参数	描述
file	必需。规定要检查的文件。
owner	规定新的所有者。可以是用户名或用户的 ID。

说明

尝试将文件 *file* 的所有者改成用户 *owner* （由用户名或用户 ID 指定）。只有超级用户可以改变文件的所有者。

例子

```
<?php
chown("test.txt","charles")
?>
```

PHP clearstatcache() 函数

定义和用法

clearstatcache() 函数清除文件状态缓存。

clearstatcache() 函数会缓存某些函数的返回信息，以便提供更高的性能。但是有时候，比如在一个脚本中多次检查同一个文件，而该文件在此脚本执行期间有被删除或修改的危险时，你需要清除文件状态缓存，以便获得正确的结果。要做到这一点，就需要使用 clearstatcache() 函数。

会进行缓存的函数，即受 clearstatcache() 函数影响的函数：

- stat()
- lstat()
- file_exists()
- is_writable()
- is_readable()
- is_executable()
- is_file()
- is_dir()
- is_link()
- filectime()
- fileatime()
- filemtime()
- fileinode()
- filegroup()
- fileowner()
- filesize()
- filetype()
- fileperms()

语法

```
clearstatcache()
```

例子

```
<?php
//检查文件大小
echo filesize("test.txt");

$file = fopen("test.txt", "a+");

// 截取文件
ftruncate($file,100);
fclose($file);

//清除缓存并再次检查文件大小
clearstatcache();
echo filesize("test.txt");
?>
```

输出：

```
792
100
```

PHP copy() 函数

定义和用法

copy() 函数拷贝文件。

语法

```
copy(_source_,_destination_)
```

参数	描述
source	必需。规定要复制的文件。
destination	必需。规定复制文件的目的地。

说明

将文件从 *source* 拷贝到 *destination*。如果成功则返回 TRUE，否则返回 FALSE。

提示和注释

提示：如果要移动文件的话，请使用 rename() 函数。

注释：从 PHP 4.3.0 开始，如果启用了 "fopen wrappers" 的话，source 和 *destination* 都可以是 URL。更多信息见 fopen()。如果 *destination* 是一个 URL，则如果封装协议不支持覆盖已有的文件时拷贝操作会失败。

重要事项：如果目标文件已存在，将会被覆盖。

例子

```
<?php
echo copy("source.txt","target.txt");
?>
```

输出：

```
1
```

PHP dirname() 函数

定义和用法

dirname() 函数返回路径中的目录部分。

语法

```
dirname(path)
```

参数	描述
path	必需。规定要检查的路径。

说明

path 参数是一个包含有指向一个文件的全路径的字符串。该函数返回去掉文件名后的目录名。

例子

```
<?php
echo dirname("c:/testweb/home.php");
echo dirname("/testweb/home.php");
?>
```

输出：

```
c:/testweb
/testweb
```

PHP disk_free_space() 函数

定义和用法

disk_free_space() 函数返回目录中的可用空间

语法

```
disk_free_space(directory)
```

参数	描述
directory	必需。规定要检查的目录。

说明

directory 参数是一个目录的字符串。该函数将根据相应的文件系统或磁盘分区返回可用的字节数。

例子

```
<?php
echo disk_free_space("C:");
?>
```

输出：

```
209693288558
```

PHP disk_total_space() 函数

定义和用法

disk_total_space() 函数返回指定目录的磁盘总大小。

语法

```
disk_total_space(directory)
```

参数	描述
directory	必需。规定要检查的目录。

说明

directory 参数是一个目录的字符串。该函数将根据相应的文件系统或磁盘分区返回所有的字节数。

提示和注释

提示：本函数返回的是该目录所在的磁盘分区的总大小，因此在给出同一个磁盘分区的不同目录作为参数所得到的结果完全相同。在 Unix 和 Windows 200x/XP 中都支持将一个磁盘分区加载为一个子目录，这时正确使用本函数就很有意义。

例子

```
<?php
echo disk_total_space("C:");
?>
```

输出类似这样：

```
509693888668
```

PHP diskfreespace() 函数

定义和用法

diskfreespace() 函数返回目录中的可用空间。该函数是 disk_free_space() 函数的别名。

语法

```
diskfreespace(directory)
```

参数	描述
directory	必需。规定要检查的目录。

说明

directory 参数是一个目录的字符串。该函数将根据相应的文件系统或磁盘分区返回可用的字节数。

例子

```
<?php
echo diskfreespace("C:");
?>
```

输出：

```
209693288558
```

PHP fclose() 函数

定义和用法

fclose() 函数关闭一个打开文件。

语法

```
fclose(file)
```

参数	描述
file	必需。规定要关闭的文件。

说明

file 参数是一个文件指针。fclose() 函数关闭该指针指向的文件。

如果成功则返回 true，否则返回 false。

文件指针必须有效，并且是通过 fopen() 或 fsockopen() 成功打开的。

例子

```
<?php
$file = fopen("test.txt","r");

//执行的一些代码...

fclose($file);
?>
```

PHP feof() 函数

定义和用法

feof() 函数检测是否已到达文件末尾 (eof)。

如果文件指针到了 EOF 或者出错时则返回 TRUE，否则返回一个错误（包括 socket 超时），其它情况则返回 FALSE。

语法

```
feof(file)
```

参数	描述
file	必需。规定要检查的打开文件。

说明

file 参数是一个文件指针。这个文件指针必须有效，并且必须指向一个由 fopen() 或 fsockopen() 成功打开（但还没有被 fclose() 关闭）的文件。

提示和注释

提示：feof() 函数对遍历长度未知的数据很有用。

注意：如果服务器没有关闭由 fsockopen() 所打开的连接，feof() 会一直等待直到超时而返回 TRUE。默认的超时限制是 60 秒，可以使用 stream_set_timeout() 来改变这个值。

注意：如果传递的文件指针无效可能会陷入无限循环中，因为 EOF 不会返回 TRUE。

例子

```
<?php
$file = fopen("test.txt", "r");

//输出文本中所有的行，直到文件结束为止。
while(! feof($file))
  {
  echo fgets($file). "<br />";
  }

fclose($file);
?>
```

输出：

```
Hello, this is a test file.
There are three lines here.
This is the last line.
```

PHP fflush() 函数

定义和用法

fflush() 函数将缓冲内容输出到文件。

语法

```
fflush(file)
```

参数	描述
file	必需。规定要检查的文件流。

说明

本函数强制将所有缓冲的输出写入 *file* 文件句柄所指向的资源。如果成功则返回 true，否则返回 false。

文件指针必须有效，并且必须指向一个由 fopen() 或 fsockopen() 成功打开（但还没有被 fclose() 关闭）的文件。

例子

```
<?php
file = fopen("test.txt","r+");

// 一些代码

fflush($file);
?>
```

PHP fgetc() 函数

定义和用法

fgetc() 函数从文件指针中读取一个字符。

语法

```
fgetc(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

返回一个包含有一个字符的字符串，该字符从 *file* 指向的文件中得到。碰到 EOF 则返回 false。

文件指针必须有效，并且必须指向一个由 fopen() 或 fsockopen() 成功打开（但还没有被 fclose() 关闭）的文件。

提示和注释

注意：该函数可能返回布尔值 false，但也可能返回一个与 false 等值的非布尔值，例如 0 或者 ""。

注释：该函数可安全用于二进制对象。

例子

例子 1

```
<?php

$file = fopen("test.txt","r");
echo fgetc($file);
fclose($file);

?>
```

输出类似：

```
H
```

例子 2

```
<?php

$file = fopen("test.txt","r");

while (! feof ($file))
  {
  echo fgetc($file);
  }

fclose($file);
?>
```

输出类似：

```
Hello, this is a test file.
```

PHP fgetcsv() 函数

定义和用法

fgetcsv() 函数从文件指针中读入一行并解析 CSV 字段。

与 fgets() 类似，不同的是 fgetcsv() 解析读入的行并找出 CSV 格式的字段，然后返回一个包含这些字段的数组。

fgetcsv() 出错时返回 FALSE，包括碰到文件结束时。

注释：从 PHP 4.3.5 起，fgetcsv() 的操作是二进制安全的。

语法

```
fgetcsv(file,length,separator,enclosure)
```

参数	描述
file	必需。规定要检查的文件。
length	可选。规定行的最大长度。必须大于 CVS 文件内最长的一行。在 PHP 5 中该参数是可选的。在 PHP 5 之前是必需的。如果忽略（在 PHP 5.0.4 以后的版本中设为 0）该参数的话，那么长度就没有限制，不过可能会影响执行效率。
separator	可选。设置字段分界符（只允许一个字符），默认值为逗号。
enclosure	可选。设置字段环绕符（只允许一个字符），默认值为双引号。该参数是在 PHP 4.3.0 中添加的。

提示和注释

注释：CSV 文件中的空行将被返回为一个包含有单个 null 字段的数组，不会被当成错误。

注释：该函数对区域设置是敏感的。比如说 LANG 设为 en_US.UTF-8 的话，单字节编码的文件就会出现读取错误。

注释：如果碰到 PHP 在读取文件时不能识别 Macintosh 文件的行结束符，可以激活 auto_detect_line_endings 运行时配置选项。

例子

例子 1

```
<?php

$file = fopen("contacts.csv","r");
print_r(fgetcsv($file));
fclose($file);

?>
```

CSV 文件：

```
George, John, Thomas, USA
James, Adrew, Martin, USA
```

输出类似：

```
Array
(
[0] => George
[1] => John
[2] => Thomas
[3] => USA
)
```

例子 2

```
<?php

$file = fopen("contacts.csv","r");

while(! feof($file))
  {
  print_r(fgetcsv($file));
  }

fclose($file);

?>
```

CSV 文件：

```
George, John, Thomas, USA
James, Adrew, Martin, USA
```

输出类似：

```
Array
(
[0] => George
[1] => John
[2] => Thomas
[3] => USA

Array
(
[0] => James
[1] => Adrew
[2] => Martin
[3] => USA
)
```

PHP fgets() 函数

定义和用法

fgets() 函数从文件指针中读取一行。

语法

```
fgets(file,length)
```

参数	描述
file	必需。规定要读取的文件。
length	可选。规定要读取的字节数。默认是 1024 字节。

说明

从 *file* 指向的文件中读取一行并返回长度最多为 *length* - 1 字节的字符串。碰到换行符（包括在返回值中）、EOF 或者已经读取了 *length* - 1 字节后停止（要看先碰到那一种情况）。如果没有指定 *length*，则默认为 1K，或者说 1024 字节。

若失败，则返回 false。

提示和注释

注释：*length* 参数从 PHP 4.2.0 起成为可选项，如果忽略，则行的长度被假定为 1024 字节。从 PHP 4.3 开始，忽略掉 *length* 将继续从流中读取数据直到行结束。如果文件中的大多数行都大于 8 KB，则在脚本中指定最大行的长度在利用资源上更为有效。

注释：从 PHP 4.3 开始本函数可以安全用于二进制文件。早期的版本则不行。

注释：如果碰到 PHP 在读取文件时不能识别 Macintosh 文件的行结束符，可以激活 auto_detect_line_endings 运行时配置选项。

例子

例子 1

```
<?php

$file = fopen("test.txt","r");
echo fgets($file);
fclose($file);

?>
```

输出类似：

```
Hello, this is a test file.
```

例子 2

```
<?php

$file = fopen("test.txt","r");

while(! feof($file))
  {
  echo fgets($file). "<br />";
  }

fclose($file);
?>
```

输出类似：

```
Hello, this is a test file.
There are three lines here.
This is the last line.
```

PHP fgetss() 函数

定义和用法

fgetss() 函数从打开的文件中读取一行并过滤掉 HTML 和 PHP 标记。

与 fgets() 相同，不同的是 fgetss 尝试从读取的文本中去掉任何 HTML 和 PHP 标记。

语法

```
fgetss(_file_,_length_,_tags_)
```

参数	描述
file	必需。规定要读取的文件。
length	可选。规定要读取的字节数。默认是 1024 字节。该参数在 PHP 5 之前是必需的。
tags	可选。规定不会被删除的标签。

说明

可以用可选的第三个参数 *tags* 指定哪些标记不被去掉。

若失败，则返回 false。

例子

例子 1

```
<?php

$file = fopen("test.htm","r");
echo fgetss($file);
fclose($file);

?>
```

输出类似：

```
This is a paragraph.
```

例子 2

```
<?php

$file = fopen("test.htm","r");
echo fgetss($file,1024,"<p>,<b>");
fclose($file);

?>
```

输出类似：

```
**This is a paragraph.**
```

输出的源代码是：

```
<p><b>This is a paragraph.</b></p>
```

PHP file() 函数

定义和用法

file() 函数把整个文件读入一个数组中。

与 file_get_contents() 类似，不同的是 file() 将文件作为一个数组返回。数组中的每个单元都是文件中相应的一行，包括换行符在内。

如果失败，则返回 false。

语法

```
file(path,include_path,context)
```

参数	描述
path	必需。规定要读取的文件。
include_path	可选。如果也想在 include_path 中搜寻文件的话，可以将该参数设为 "1"。
context	可选。规定文件句柄的环境。context 是一套可以修改流的行为的选项。若使用 null，则忽略。

说明

对 *context* 的支持是 PHP 5.0.0 添加的。

返回的数组中每一行都包括了行结束符，因此如果不需要行结束符时还需要使用 rtrim() 函数。

提示和注释

注释：从 PHP 4.3.0 开始，可以用 file_get_contents() 来将文件读入到一个字符串并返回。

注释：从 PHP 4.3.0 开始，file() 可以安全用于二进制文件。

注释：如果碰到 PHP 在读取文件时不能识别 Macintosh 文件的行结束符，可以激活 auto_detect_line_endings 运行时配置选项。

例子

```
<?php
print_r(file("test.txt"));
?>
```

输出：

```
Array
(
[0] => Hello World. Testing testing!
[1] => Another day, another line.
[2] => If the array picks up this line,
[3] => then is it a pickup line?
)
```

PHP file_exists() 函数

定义和用法

file_exists() 函数检查文件或目录是否存在。

如果指定的文件或目录存在则返回 true，否则返回 false。

语法

```
file_exists(path)
```

参数	描述
path	必需。规定要检查的路径。

例子

```
<?php
echo file_exists("test.txt");
?>
```

输出：

```
1
```

PHP file_get_contents() 函数

定义和用法

file_get_contents() 函数把整个文件读入一个字符串中。

和 file() 一样，不同的是 file_get_contents() 把文件读入一个字符串。

file_get_contents() 函数是用于将文件的内容读入到一个字符串中的首选方法。如果操作系统支持，还会使用内存映射技术来增强性能。

语法

```
file_get_contents(_path_,_include_path_,_context_,_start_,_max_length_)
```

参数	描述
path	必需。规定要读取的文件。
include_path	可选。如果也想在 include_path 中搜寻文件的话，可以将该参数设为 "1"。
context	可选。规定文件句柄的环境。context 是一套可以修改流的行为的选项。若使用 null，则忽略。
start	可选。规定在文件中开始读取的位置。该参数是 PHP 5.1 新加的。
max_length	可选。规定读取的字节数。该参数是 PHP 5.1 新加的。

说明

对 *context* 参数的支持是 PHP 5.0.0 添加的。

提示和注释

注释：本函数可安全用于二进制对象。

例子

```
<?php
echo file_get_contents("test.txt");
?>
```

输出：

```
This is a test file with test text.
```

PHP file_put_contents() 函数

定义和用法

file_put_contents() 函数把一个字符串写入文件中。

与依次调用 fopen()，fwrite() 以及 fclose() 功能一样。

语法

```
file_put_contents(file,data,mode,context)
```

参数	描述
file	必需。规定要写入数据的文件。如果文件不存在，则创建一个新文件。
data	可选。规定要写入文件的数据。可以是字符串、数组或数据流。
mode	可选。规定如何打开/写入文件。可能的值：`FILE_USE_INCLUDE_PATH` `FILE_APPEND` `LOCK_EX`
context	可选。规定文件句柄的环境。context 是一套可以修改流的行为的选项。若使用 null，则忽略。

说明

参数 *data* 可以是数组（但不能是多维数组）。

自 PHP 5.1.0 起，*data* 参数也可以被指定为 stream 资源，stream 中所保存的缓存数据将被写入到指定文件中，这种用法就相似于使用 stream_copy_to_stream() 函数。

对 *context* 参数的支持是 PHP 5.0.0 添加的。

返回值

该函数将返回写入到文件内数据的字节数。

提示和注释

提示：使用 FILE_APPEND 可避免删除文件中已有的内容。

注释：本函数可安全用于二进制对象。

例子

```
<?php
echo file_put_contents("test.txt","Hello World. Testing!");
?>
```

输出：

```
26
```

PHP fileatime() 函数

定义和用法

fileatime() 函数返回指定文件的上次访问时间。

该函数返回文件上次被访问的时间。如果出错则返回 false。时间以 Unix 时间戳的方式返回。

语法

```
fileatime(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

注释：文件的 atime 应该在不论何时读取了该文件中的数据块时被更改。当一个应用程序定期访问大量文件或目录时很影响性能。有些 Unix 文件系统可以在加载时关闭 atime 的更新以提高这类程序的性能。USENET 新闻组假脱机是一个常见的例子。在这种文件系统下，本函数没有用处。

例子

```
<?php
echo fileatime("test.txt");
echo "Last access: ".date("F d Y H:i:s.",fileatime("test.txt"));
?>
```

输出：

```
1140684501
Last access: February 23 2006 09:48:21.
```

PHP filectime() 函数

定义和用法

filectime() 函数返回指定文件的上次 inode 修改时间。

该函数返回文件上次 inode 被修改的时间。如果出错则返回 false。时间以 Unix 时间戳的方式返回。

语法

```
fileatime(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

注意：在大多数 Unix 文件系统中，当一个文件的 inode 数据被改变时则该文件被认为是修改了。也就是说，当文件的权限，所有者，所有组或其它 inode 中的元数据被更新时。参见 filemtime()（这才是你想用于在 Web 页面中建立"最后更新时间"脚注的函数）和 fileatime()。

注释：某些 Unix 说明文本中把 ctime 说成是该文件建立的时间，这是错的。在大多数 Unix 文件系统中，没有 Unix 文件的建立时间。

例子

```
<?php
echo filectime("test.txt");
echo "Last change: ".date("F d Y H:i:s.",filectime("test.txt"));
?>
```

输出：

```
1138609592
Last change: January 30 2006 09:26:32.
```

PHP filegroup() 函数

定义和用法

filegroup() 函数返回指定文件的组 ID。

若成功，则返回指定文件所属组的 ID。若失败，则返回 false 以及一个 E_WARNING 级别的错误。

组 ID 以数字格式返回。

语法

```
filegroup(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

提示：请使用 posix_getgrgid() 来将组 ID 转换为组名。

例子

```
<?php
echo filegroup("test.txt");
?>
```

PHP fileinode() 函数

定义和用法

fileinode() 函数返回文件的 inode 编号。

若成功，则返回指定文件的 inode 节点号。若失败，则返回 false。

语法

```
fileinode(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
echo fileinode("test.txt");
?>
```

PHP filemtime() 函数

定义和用法

filemtime() 函数返回文件内容上次的修改时间。

若成功，则时间以 Unix 时间戳的方式返回。若失败，则返回 false。

语法

```
filemtime(filename)
```

参数	描述
filename	必需。规定要检查的文件。

说明

本函数返回文件中的数据块上次被写入的时间，也就是说，文件的内容上次被修改的时间。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
echo filemtime("test.txt");
echo "Last modified: ".date("F d Y H:i:s.",filemtime("test.txt"));
?>
```

输出：

```
1139919766
Last modified: February 14 2006 13:22:46.
```

PHP fileowner() 函数

定义和用法

fileowner() 函数返回文件的所有者。

若成功，则返回文件所有的用户 ID。若失败，则返回 false。用户 ID 以数字格式返回。

语法

```
fileowner(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

提示：用户 ID 以数字格式返回，请使用 posix_getpwuid() 来把用户 ID 转换为用户名。

例子

```
<?php
echo fileowner("test.txt");
?>
```

PHP fileperms() 函数

定义和用法

fileperms() 函数返回文件或目录的权限。

若成功，则返回文件的访问权限。若失败，则返回 false。

语法

```
fileperms(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

例子 1

```
<?php
echo fileperms("test.txt");
?>
```

输出：

```
33206
```

例子 2

以八进制值返回权限：

```
<?php
echo substr(sprintf("%o",fileperms("test.txt")),-4);
?>
```

输出：

```
1777
```

PHP filesize() 函数

定义和用法

filesize() 函数返回指定文件的大小。

若成功，则返回文件大小的字节数。若失败，则返回 false 并生成一条 E_WARNING 级的错误。

语法

```
filesize(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
echo filesize("test.txt");
?>
```

输出：

```
20
```

PHP filetype() 函数

定义和用法

filetype() 函数返回指定文件或目录的类型。

若成功，则返回 7 种可能的值。若失败，则返回 false。

可能的值：

- fifo
- char
- dir
- block
- link
- file
- unknown

语法

```
filetype(filename)
```

参数	描述
filename	必需。规定要检查的文件。

提示和注释

提示：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

例子 1

```
<?php
echo filetype("test.txt");
?>
```

输出：

```
file
```

例子 2

```
<?php
echo filetype("images");
?>
```

输出：

```
dir
```

PHP flock() 函数

定义和用法

flock() 函数锁定或释放文件。

若成功，则返回 true。若失败，则返回 false。

语法

```
flock(file,lock,block)
```

参数	描述
file	必需。规定要锁定或释放的已打开的文件。
lock	必需。规定要使用哪种锁定类型。
block	可选。若设置为 1 或 true，则当进行锁定时阻挡其他进程。

说明

flock() 操作的 *file* 必须是一个已经打开的文件指针。

lock 参数可以是以下值之一：

- 要取得共享锁定（读取的程序），将 *lock* 设为 LOCK_SH（PHP 4.0.1 以前的版本设置为 1）。
- 要取得独占锁定（写入的程序），将 *lock* 设为 LOCK_EX（PHP 4.0.1 以前的版本中设置为 2）。
- 要释放锁定（无论共享或独占），将 *lock* 设为 LOCK_UN（PHP 4.0.1 以前的版本中设置为 3）。
- 如果不希望 flock() 在锁定时堵塞，则给 *lock* 加上 LOCK_NB（PHP 4.0.1 以前的版本中设置为 4）。

提示和注释

提示：可以通过 fclose() 来释放锁定操作，代码执行完毕时也会自动调用。

注释：由于 flock() 需要一个文件指针， 因此可能不得不用一个特殊的锁定文件来保护打算通过写模式打开的文件的访问（在 fopen() 函数中加入 "w" 或 "w+"）。

例子

```
<?php

$file = fopen("test.txt","w+");

// 排它性的锁定
if (flock($file,LOCK_EX))
  {
  fwrite($file,"Write something");
  // release lock
  flock($file,LOCK_UN);
  }
else
  {
  echo "Error locking file!";
  }

fclose($file);
?>
```

PHP fnmatch() 函数

定义和用法

fnmatch() 函数根据指定的模式来匹配文件名或字符串。

语法

```
fnmatch(pattern,string,flags)
```

参数	描述
pattern	必需。规定要检索的模式。
string	必需。规定要检查的字符串或文件。
flags	可选。

说明

此函数对于文件名尤其有用，但也可以用于普通的字符串。普通用户可能习惯于 shell 模式或者至少其中最简单的形式 '?' 和 '*' 通配符，因此使用 fnmatch() 来代替 ereg() 或者 preg_match() 来进行前端搜索表达式输入对于非程序员用户更加方便。

提示和注释

重要事项：目前该函数无法在 Windows 或其它非 POSIX 兼容的系统上使用。

例子

根据 shell 通配符来检查颜色名：

```
<?php
$txt = "My car is darkgrey..."
if (fnmatch("*gr[ae]y",$txt))
  {
  echo "some form of gray ...";
  }
?>
```

PHP fopen() 函数

定义和用法

fopen() 函数打开文件或者 URL。

如果打开失败，本函数返回 FALSE。

语法

```
fopen(filename,mode,include_path,context)
```

参数	描述
filename	必需。规定要打开的文件或 URL。
mode	必需。规定要求到该文件/流的访问类型。可能的值见下表。
include_path	可选。如果也需要在 include_path 中检索文件的话，可以将该参数设为 1 或 TRUE。
context	可选。规定文件句柄的环境。Context 是可以修改流的行为的一套选项。

mode 参数的可能的值

mode	说明
"r"	只读方式打开，将文件指针指向文件头。
"r+"	读写方式打开，将文件指针指向文件头。
"w"	写入方式打开，将文件指针指向文件头并将文件大小截为零。如果文件不存在则尝试创建之。
"w+"	读写方式打开，将文件指针指向文件头并将文件大小截为零。如果文件不存在则尝试创建之。
"a"	写入方式打开，将文件指针指向文件末尾。如果文件不存在则尝试创建之。
"a+"	读写方式打开，将文件指针指向文件末尾。如果文件不存在则尝试创建之。
"x"	创建并以写入方式打开，将文件指针指向文件头。如果文件已存在，则 fopen() 调用失败并返回 FALSE，并生成一条 E_WARNING 级别的错误信息。如果文件不存在则尝试创建之。这和给底层的 open(2) 系统调用指定 O_EXCL|O_CREAT 标记是等价的。此选项被 PHP 4.3.2 以及以后的版本所支持，仅能用于本地文件。
"x+"	创建并以读写方式打开，将文件指针指向文件头。如果文件已存在，则 fopen() 调用失败并返回 FALSE，并生成一条 E_WARNING 级别的错误信息。如果文件不存在则尝试创建之。这和给底层的 open(2) 系统调用指定 O_EXCL|O_CREAT 标记是等价的。此选项被 PHP 4.3.2 以及以后的版本所支持，仅能用于本地文件。

说明

fopen() 将 *filename* 指定的名字资源绑定到一个流上。如果 *filename* 是 "*scheme://...*" 的格式，则被当成一个 URL，PHP 将搜索协议处理器（也被称为封装协议）来处理此模式。如果该协议尚未注册封装协议，PHP 将发出一条消息来帮助检查脚本中潜在的问题并将 *filename* 当成一个普通的文件名继续执行下去。

如果 PHP 认为 *filename* 指定的是一个本地文件，将尝试在该文件上打开一个流。该文件必须是 PHP 可以访问的，因此需要确认文件访问权限允许该访问。如果激活了安全模式或者 open_basedir 则会应用进一步的限制。

如果 PHP 认为 *filename* 指定的是一个已注册的协议，而该协议被注册为一个网络 URL，PHP 将检查并确认 allow_url_fopen 已被激活。如果关闭了，PHP 将发出一个警告，而 fopen 的调用则失败。

对 *context* 的支持是 PHP 5.0.0 添加的。

提示和注释

注释：不同的操作系统家族具有不同的行结束习惯。当写入一个文本文件并想插入一个新行时，需要使用符合操作系统的行结束符号。基于 Unix 的系统使用 \n 作为行结束字符，基于 Windows 的系统使用 \r\n 作为行结束字符，基于 Macintosh 的系统使用 \r 作为行结束字符。如果写入文件时使用了错误的行结束符号，则其它应用程序打开这些文件时可能会表现得很怪异。

Windows 下提供了一个文本转换标记（"t"）可以透明地将 \n 转换为 \r\n。与此对应还可以使用 "b" 来强制使用二进制模式，这样就不会转换数据。要使用这些标记，要么用 "b" 或者用 "t" 作为 mode 参数的最后一个字符。

默认的转换模式依赖于 SAPI 和所使用的 PHP 版本，因此为了便于移植鼓励总是指定恰当的标记。如果是操作纯文本文件并在脚本中使用了 \n 作为行结束符，但还要期望这些文件可以被其它应用程序例如 Notepad 读取，则在 mode 中使用 "t"。在所有其它情况下使用 "b"。

在操作二进制文件时如果没有指定 "b" 标记，可能会碰到一些奇怪的问题，包括坏掉的图片文件以及关于 \r\n 字符的奇怪问题。

注释：为移植性考虑，强烈建议在用 fopen() 打开文件时总是使用 "b" 标记。

注释：再一次，为移植性考虑，强烈建议你重写那些依赖于 "t" 模式的代码使其使用正确的行结束符并改成 "b" 模式。

例子

```
<?php
$file = fopen("test.txt","r");
$file = fopen("/home/test/test.txt","r");
$file = fopen("/home/test/test.gif","wb");
$file = fopen("http://www.example.com/","r");
$file = fopen("ftp://user:password@example.com/test.txt","w");
?>
```

PHP fpassthru() 函数

定义和用法

fpassthru() 函数输出文件指针处的所有剩余数据。

该函数将给定的文件指针从当前的位置读取到 EOF，并把结果写到输出缓冲区。

语法

```
fpassthru(file)
```

参数	描述
file	必需。规定要读取的打开文件或资源。

说明

如果发生错误， fpassthru() 返回 false。否则 fpassthru() 返回从 *file* 读取并传递到输出的字符数目。

文件指针必须有效，并且必须指向一个由 fopen() 或 fsockopen() 成功打开（但还没有被 fclose() 关闭）的文件。

提示和注释

提示：如果已经向文件写入数据，就必须调用 rewind() 来将文件指针指向文件头。

提示：如果既不修改文件也不在特定位置检索，只想将文件的内容下载到输出缓冲区，应该使用 readfile()，这样可以省去 fopen() 调用。

注释：当在 Windows 系统中将 fpassthru() 用于二进制文件时，要确保在用 fopen() 打开文件时在 mode 中附加了 b 来将文件以二进制方式打开。鼓励在处理二进制文件时使用 b 标志，即使系统并不需要，这样可以使脚本的移植性更好。

例子

例子 1

```
<?php
$file = fopen("test.txt","r");

// 读取第一行
fgets($file);

// 把文件的其余部分发送到输出缓存
echo fpassthru($file);

fclose($file);
?>
```

输出：

```
There are three lines in this file.
This is the last line.59
```

注：59 指示被传递的字符数。

例子 2

转储 www 服务器的 index 页：

```
<?php
$file = fopen("http://www.example.com","r");
fpassthru($file);
?>
```

PHP fputcsv() 函数

定义和用法

fputcsv() 函数将行格式化为 CSV 并写入一个打开的文件。

该函数返回写入字符串的长度。若出错，则返回 false。。

语法

```
fputcsv(file,fields,seperator,enclosure)
```

参数	描述
file	必需。规定要写入的打开文件。
fields	必需。规定要从中获得数据的数组。
seperator	可选。规定字段分隔符的字符。默认是逗号 (,)。
enclosure	可选。规定字段环绕符的字符。默认是双引号 "。

说明

fputcsv() 将一行（用 *fields* 数组传递）格式化为 CSV 格式并写入由 *file* 指定的文件。

提示和注释

提示：参见 fgetcsv() 函数。

例子

```
<?php
$list = array
(
"George,John,Thomas,USA",
"James,Adrew,Martin,USA",
);

$file = fopen("contacts.csv","w");

foreach ($list as $line)
  {
  fputcsv($file,split(',',$line));
  }

fclose($file);
?>
```

以上代码执行后，CSV 文件会类似这样：

```
George,John,Thomas,USA
James,Adrew,Martin,USA
```

PHP fputs() 函数

定义和用法

fputs() 函数写入文件（可安全用于二进制文件）。

fputs() 函数是 fwrite() 函数的别名。

语法

```
fputs(file,string,length)
```

参数	描述
file	必需。规定要写入的打开文件。
string	必需。规定要写入文件的字符串。
length	可选。规定要写入的最大字节数。

说明

fwrite() 把 *string* 的内容写入文件指针 *file* 处。 如果指定了 *length*，当写入了 *length* 个字节或者写完了 *string* 以后，写入就会停止，视乎先碰到哪种情况。

fwrite() 返回写入的字符数，出现错误时则返回 false。

例子

```
<?php
$file = fopen("test.txt","w");
echo fputs($file,"Hello World. Testing!");
fclose($file);
?>
```

输出：

```
21
```

PHP fread() 函数

定义和用法

fread() 函数读取文件（可安全用于二进制文件）。

语法

```
fread(file,length)
```

参数	描述
file	必需。规定要读取打开文件。
length	必需。规定要读取的最大字节数。

说明

fread() 从文件指针 *file* 读取最多 *length* 个字节。该函数在读取完最多 *length* 个字节数，或到达 EOF 的时候，或（对于网络流）当一个包可用时，或（在打开用户空间流之后）已读取了 8192 个字节时就会停止读取文件，视乎先碰到哪种情况。

返回所读取的字符串，如果出错返回 false。

提示和注释

提示：如果只是想将一个文件的内容读入到一个字符串中，请使用 file_get_contents()，它的性能比 fread() 好得多。

例子

例子 1

从文件中读取 10 个字节：

```
<?php
$file = fopen("test.txt","r");
fread($file,"10");
fclose($file);
?>
```

例子 2

读取整个文件：

```
<?php
$file = fopen("test.txt","r");
fread($file,filesize("test.txt"));
fclose($file);
?>
```

PHP fscanf() 函数

定义和用法

fscanf() 函数根据指定的格式对来自打开的文件的输入进行解析。

语法

```
fscanf(file,format,mixed)
```

参数	描述
file	必需。规定要检查的文件。
format	必需。规定格式。
mixed	可选。

说明

fscanf() 函数与 sscanf() 相似，但是它从与 *file* 关联的文件中接受输入并根据指定的 *format* 来解释输入。如果只给此函数传递了两个参数，解析后的值会被作为数组返回。否则，如果提供了可选参数，此函数将返回被赋值的数目。可选参数必须用引用传递。

提示和注释

注释：格式字符串中的任何空白会与输入流中的任何空白匹配。这意味着甚至格式字符串中的制表符 \t 也会与输入流中的一个空格字符匹配。

注释：在 PHP 4.3.0 之前，从文件中读入的最大字符数是 512（或者第一个 \n，看先碰到哪种情况）。从 PHP 4.3.0 起可以读取任意长的行。

PHP fseek() 函数

定义和用法

fseek() 函数在打开的文件中定位。

该函数把文件指针从当前位置向前或向后移动到新的位置，新位置从文件头开始以字节数度量。

成功则返回 0；否则返回 -1。注意，移动到 EOF 之后的位置不会产生错误。

语法

```
fseek(file,offset,whence)
```

参数	描述
file	必需。规定要在其中定位的文件。
offset	必需。规定新的位置（从文件头开始以字节数度量）。
whence	可选。可能的值： `SEEK_SET` - 设定位置等于 *offset* 字节。默认。 `SEEK_CUR` - 设定位置为当前位置加上 *offset*。 `SEEK_END` - 设定位置为文件末尾加上 *offset* （要移动到文件尾之前的位置，*offset* 必须是一个负值）。

说明

whence 参数是 PHP 4.0.0 之后增加的。

提示和注释

提示：通过使用 ftell() 来找到当前位置。

例子

```
<?php
$file = fopen("test.txt","r");

// 读取第一行
fgets($file);

// 倒回文件的开头
fseek($file,0);
?>
```

PHP fstat() 函数

定义和用法

fstat() 函数返回关于打开文件的信息。

语法

```
fstat(file)
```

参数	描述
pipe	必需。规定要检查的打开文件。

说明

获取由文件指针 handle 所打开文件的统计信息。

该函数返回的数组具有该文件的统计信息，该数组包含以下元素：

数字下标	关联键名（自 **PHP 4.0.6**）	说明
0	dev	设备名
1	ino	号码
2	mode	inode 保护模式
3	nlink	被连接数目
4	uid	所有者的用户 id
5	gid	所有者的组 id
6	rdev	设备类型，如果是 inode 设备的话
7	size	文件大小的字节数
8	atime	上次访问时间（Unix 时间戳）
9	mtime	上次修改时间（Unix 时间戳）
10	ctime	上次改变时间（Unix 时间戳）
11	blksize	文件系统 IO 的块大小
12	blocks	所占据块的数目

提示和注释

提示：本函数与 stat() 函数相似，不同的是，它是作用于已打开的文件指针而不是文件名。

例子

```
<?php
$file = fopen("test.txt","r");
print_r(fstat($file));
fclose($file);
?>
```

输出类似：

```
Array
(
[0] => 0
[1] => 0
[2] => 33206
[3] => 1
[4] => 0
[5] => 0
[6] => 0
[7] => 92
[8] => 1141633430
[9] => 1141298003
[10] => 1138609592
[11] => -1
[12] => -1
[dev] => 0
[ino] => 0
[mode] => 33206
[nlink] => 1
[uid] => 0
[gid] => 0
[rdev] => 0
[size] => 92
[atime] => 1141633430
[mtime] => 1141298003
[ctime] => 1138609592
[blksize] => -1
[blocks] => -1
)
```

PHP ftell() 函数

定义和用法

ftell() 函数在打开文件中的当前位置。

该函数返回文件指针的当前位置。若失败，则返回 false。

语法

```
ftell(file)
```

参数	描述
file	必需。规定要检查的已打开文件。

说明

文件指针 *file* 必须是有效的，且必须指向一个通过 fopen() 或 popen() 成功打开的文件。

在附加模式（加参数 "a" 打开文件）中 ftell() 会返回未定义错误。

例子

```
<?php
$file = fopen("test.txt","r");

// 输出当前位置
echo ftell($file);

// 改变当前位置
fseek($file,"15");

// 再次输出当前位置
echo ftell($file);

fclose($file);
?>
```

输出：

```
0
15
```

PHP ftruncate() 函数

定义和用法

ftruncate() 函数把文件截断到指定的长度。

语法

```
ftruncate(file,size)
```

参数	描述
file	必需。规定要截断的打开文件。
size	必需。规定新的文件大小。

说明

接受文件指针 *file* 作为参数，并将文件大小截取为 *size*。如果成功则返回 TRUE，否则返回 FALSE。

提示和注释

注释：文件只会在 append 模式下改变。在 write 模式下，必须加上 fseek() 操作。

注释：在 PHP 4.3.3 之前，ftruncate() 在成功时返回一个整数值 1，而不是布尔值的 TRUE。

例子

```
<?php
//检查文件大小
echo filesize("test.txt");
echo "<br />";

$file = fopen("test.txt", "a+");
ftruncate($file,100);
fclose($file);

//清空缓存，再次检查文件大小
clearstatcache();
echo filesize("test.txt");
?>
```

输出类似：

```
792
100
```

PHP fwrite() 函数

定义和用法

fwrite() 函数写入文件（可安全用于二进制文件）。

语法

```
fwrite(file,string,length)
```

参数	描述
file	必需。规定要写入的打开文件。
string	必需。规定要写入文件的字符串。
length	可选。规定要写入的最大字节数。

说明

fwrite() 把 *string* 的内容写入文件指针 *file* 处。 如果指定了 *length*，当写入了 *length* 个字节或者写完了 *string* 以后，写入就会停止，视乎先碰到哪种情况。

fwrite() 返回写入的字符数，出现错误时则返回 false。

例子

```
<?php
$file = fopen("test.txt","w");
echo fwrite($file,"Hello World. Testing!");
fclose($file);
?>
```

输出：

```
21
```

PHP glob() 函数

定义和用法

glob() 函数返回匹配指定模式的文件名或目录。

该函数返回一个包含有匹配文件 / 目录的数组。如果出错返回 false。

语法

```
glob(pattern,flags)
```

参数	描述
file	必需。规定检索模式。
size	可选。规定特殊的设定。 GLOB_MARK - 在每个返回的项目中加一个斜线 GLOB_NOSORT - 按照文件在目录中出现的原始顺序返回（不排序） GLOB_NOCHECK - 如果没有文件匹配则返回用于搜索的模式 GLOB_NOESCAPE - 反斜线不转义元字符 GLOB_BRACE - 扩充 {a,b,c} 来匹配 'a'，'b' 或 'c' GLOB_ONLYDIR - 仅返回与模式匹配的目录项 GLOB_ERR - 停止并读取错误信息（比如说不可读的目录），默认的情况下忽略所有错误 注释：GLOB_ERR 是 PHP 5.1 添加的。

例子

例子 1

```
<?php
print_r(glob("*.txt"));
?>
```

输出类似：

```
Array
(
[0] => target.txt
[1] => source.txt
[2] => test.txt
[3] => test2.txt
)
```

例子 2

```
<?php
print_r(glob("*.*"));
?>
```

输出类似：

```
Array
(
[0] => contacts.csv
[1] => default.php
[2] => target.txt
[3] => source.txt
[4] => tem1.tmp
[5] => test.htm
[6] => test.ini
[7] => test.php
[8] => test.txt
[9] => test2.txt
)
```

PHP is_dir() 函数

定义和用法

is_dir() 函数检查指定的文件是否是目录。

语法

```
is_dir(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果文件名存在并且为目录，则返回 true。如果 *file* 是一个相对路径，则按照当前工作目录检查其相对路径。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
$file = "images";
if(is_dir($file))
  {
  echo ("$file is a directory");
  }
else
  {
  echo ("$file is not a directory");
  }
?>
```

输出：

```
images is a directory
```

PHP is_executable() 函数

定义和用法

is_executable() 函数检查指定的文件是否可执行。

语法

```
is_executable(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果文件存在且可执行，则返回 true。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

注释：is_executable() 自 PHP 5.0.0 版起可用于 Windows。

例子

```
<?php
$file = "setup.exe";
if(is_executable($file))
  {
  echo ("$file is executable");
  }
else
  {
  echo ("$file is not executable");
  }
?>
```

输出：

```
setup.exe is executable
```

PHP is_file() 函数

定义和用法

is_file() 函数检查指定的文件名是否是正常的文件。

语法

```
is_file(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果文件存在且为正常的文件，则返回 true。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

例子 1

```
<?php
$file = "test.txt";
if(is_file($file))
  {
  echo ("$file is a regular file");
  }
else
  {
  echo ("$file is not a regular file");
  }
?>
```

输出：

```
test.txt is a regular file
```

例子 2

```
<?php
var_dump(is_file('a_file.txt')) . "\n";
var_dump(is_file('/usr/bin/')) . "\n";
?>
```

输出：

```
bool(true)
bool(false)
```

PHP is_link() 函数

定义和用法

is_link() 函数判断指定文件名是否为一个符号连接。

语法

```
is_link(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果文件存在并且是一个符号连接，则返回 true。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
$link = "images";
if(is_link($link))
  {
  echo ("$link is a link");
  }
else
  {
  echo ("$link is not a link");
  }
?>
```

输出：

```
images is not a link
```

PHP is_readable() 函数

定义和用法

is_readable() 函数判断指定文件名是否可读。

语法

```
is_readable(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果由 *file* 指定的文件或目录存在并且可读，则返回 TRUE。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
$file = "test.txt";
if(is_readable($file))
  {
  echo ("$file is readable");
  }
else
  {
  echo ("$file is not readable");
  }
?>
```

输出：

```
test.txt is readable
```

PHP is_uploaded_file() 函数

定义和用法

is_uploaded_file() 函数判断指定的文件是否是通过 HTTP POST 上传的。

语法

```
is_uploaded_file(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果 *file* 所给出的文件是通过 HTTP POST 上传的则返回 TRUE。

该函数可以用于确保恶意的用户无法欺骗脚本去访问本不能访问的文件，例如 /etc/passwd。

这种检查显得格外重要，如果上传的文件有可能会造成对用户或本系统的其他用户显示其内容的话。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
$file = "test.txt";
if(is_uploaded_file($file))
  {
  echo ("$file is uploaded via HTTP POST");
  }
else
  {
  echo ("$file is not uploaded via HTTP POST");
  }
?>
```

输出：

```
test.txt is not uploaded via HTTP POST
```

PHP is_writable() 函数

定义和用法

is_writable() 函数判断指定的文件是否可写。

语法

```
is_writable(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果文件存在并且可写则返回 true。*file* 参数可以是一个允许进行是否可写检查的目录名。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
$file = "test.txt";
if(is_writable($file))
  {
  echo ("$file is writeable");
  }
else
  {
  echo ("$file is not writeable");
  }
?>
```

输出：

```
test.txt is writeable
```

PHP is_writeable() 函数

定义和用法

is_writeable() 函数判断指定的文件是否可写。

该函数是 is_writable() 函数的别名。

语法

```
is_writeable(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

如果文件存在并且可写则返回 true。*file* 参数可以是一个允许进行是否可写检查的目录名。

提示和注释

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
$file = "test.txt";
if(is_writeable($file))
  {
  echo ("$file is writeable");
  }
else
  {
  echo ("$file is not writeable");
  }
?>
```

输出：

```
test.txt is writeable
```

PHP link() 函数

定义和用法

link() 函数建立一个硬连接。

如果成功，则返回 true，失败则返回 false。

提示：创建的连接不是 HTML 链接，而是文件系统中的连接。

语法

```
link(target,link)
```

参数	描述
target	必需。
link	必需。

提示和注释

注释：本函数不能作用于远程文件，被检查的文件必须通过服务器的文件系统访问。

注释：本函数不能工作在 windows 平台上。

PHP linkinfo() 函数

定义和用法

linkinfo() 函数返回连接的信息。

本函数返回设备 ID。若出错，则返回 0 或 FALSE。

语法

```
linkinfo(path)
```

参数	描述
path	必需。规定要检查的路径

提示和注释

注释：本函数不能工作在 windows 平台上。

PHP lstat() 函数

定义和用法

lstat() 函数返回关于文件或符号连接的信息。

语法

```
lstat(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

获取由 *file* 参数指定的文件或符号连接的统计信息。

lstat() 的返回格式

数字下标	关联键名（自 **PHP 4.0.6**）	说明
0	dev	设备名
1	ino	号码
2	mode	inode 保护模式
3	nlink	被连接数目
4	uid	所有者的用户 id
5	gid	所有者的组 id
6	rdev	设备类型，如果是 inode 设备的话
7	size	文件大小的字节数
8	atime	上次访问时间（Unix 时间戳）
9	mtime	上次修改时间（Unix 时间戳）
10	ctime	上次改变时间（Unix 时间戳）
11	blksize	文件系统 IO 的块大小
12	blocks	所占据块的数目

提示和注释

提示：本函数与 stat() 函数相同，不同之处只有一点：如果 *file* 参数是符号连接的话，则该符号连接的状态被返回，而不是该符号连接所指向的文件的状态。

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
print_r(lstat("test.txt"));
?>
```

输出类似：

```
Array
(
[0] => 0
[1] => 0
[2] => 33206
[3] => 1
[4] => 0
[5] => 0
[6] => 0
[7] => 92
[8] => 1141633430
[9] => 1141298003
[10] => 1138609592
[11] => -1
[12] => -1
[dev] => 0
[ino] => 0
[mode] => 33206
[nlink] => 1
[uid] => 0
[gid] => 0
[rdev] => 0
[size] => 92
[atime] => 1141633430
[mtime] => 1141298003
[ctime] => 1138609592
[blksize] => -1
[blocks] => -1
)
```

PHP mkdir() 函数

定义和用法

mkdir() 函数创建目录。

若成功，则返回 true，否则返回 false。

语法

```
mkdir(path,mode,recursive,context)
```

参数	描述
path	必需。规定要创建的目录的名称。
mode	必需。规定权限。默认是 0777。
recursive	必需。规定是否设置递归模式。
context	必需。规定文件句柄的环境。Context 是可修改流的行为的一套选项。

说明

mkdir() 尝试新建一个由 *path* 指定的目录。

默认的 *mode* 是 0777，意味着最大可能的访问权。

提示和注释

注释：*mode* 在 Windows 下被忽略。自 PHP 4.2.0 起成为可选项。

注释：对 *context* 的支持是 PHP 5.0.0 添加的。

注释：*recursive* 参数是 PHP 5.0.0 添加的。

例子

```
<?php
mkdir("testing");
?>
```

PHP move_uploaded_file() 函数

定义和用法

move_uploaded_file() 函数将上传的文件移动到新位置。

若成功，则返回 true，否则返回 false。

语法

```
move_uploaded_file(file,newloc)
```

参数	描述
file	必需。规定要移动的文件。
newloc	必需。规定文件的新位置。

说明

本函数检查并确保由 *file* 指定的文件是合法的上传文件（即通过 PHP 的 HTTP POST 上传机制所上传的）。如果文件合法，则将其移动为由 *newloc* 指定的文件。

如果 *file* 不是合法的上传文件，不会出现任何操作，move_uploaded_file() 将返回 false。

如果 *file* 是合法的上传文件，但出于某些原因无法移动，不会出现任何操作，move_uploaded_file() 将返回 false，此外还会发出一条警告。

这种检查显得格外重要，如果上传的文件有可能会造成对用户或本系统的其他用户显示其内容的话。

提示和注释

注释：本函数仅用于通过 HTTP POST 上传的文件。

注意：如果目标文件已经存在，将会被覆盖。

PHP parse_ini_file() 函数

定义和用法

parse_ini_file() 函数解析一个配置文件，并以数组的形式返回其中的设置。

语法

```
parse_ini_file(file,process_sections)
```

参数	描述
file	必需。规定要检查的 ini 文件。
process_sections	可选。如果设置为 true，则返回一个多维数组，包括了配置文件中每一节的名称和设置。默认是 false。

说明

ini 文件的结构和 php.ini 的相似。

常量也可以在 ini 文件中被解析，因此如果在运行 parse_ini_file() 之前定义了常量作为 ini 的值，将会被集成到结果中去。只有 ini 的值会被求值。

由数字组成的键名和小节名会被 PHP 当作整数来处理，因此以 0 开头的数字会被当作八进制而以 0x 开头的会被当作十六进制。

提示和注释

注释：本函数可以用来读取你自己的应用程序的配置文件。本函数与 php.ini 文件没有关系，该文件在运行脚本时就已经处理过了。

注释：如果 ini 文件中的值包含任何非字母数字的字符，需要将其括在双引号中（"）。

注释：有些保留字不能作为 ini 文件中的键名，包括：null，yes，no，true 和 false。值为 null，no 和 false 等效于 ""，值为 yes 和 true 等效于 "1"。字符 {}|"~![()" 也不能用在键名的任何地方，而且这些字符在选项值中有着特殊的意义。

注释：自 PHP 5.0 版本开始，该函数也处理选项值内的新行。

例子

例子 1

"test.ini" 的内容：

```
[names]
me = Robert
you = Peter

[urls]
first = "http://www.example.com"
second = "http://www.w3school.com.cn"
```

PHP 代码：

```
<?php
print_r(parse_ini_file("test.ini"));
?>
```

输出：

```
Array
(
[me] => Robert
[you] => Peter
[first] => http://www.example.com
[second] => http://www.w3school.com.cn
)
```

例子 2

"test.ini" 的内容：

```
[names]
me = Robert
you = Peter

[urls]
first = "http://www.example.com"
second = "http://www.w3school.com.cn"
```

PHP 代码（*process_sections* 设置为 true）：

```
<?php
print_r(parse_ini_file("test.ini",true));
?>
```

输出：

```
Array
(
[names] => Array
  (
  [me] => Robert
  [you] => Peter
  )
[urls] => Array
  (
  [first] => http://www.example.com
  [second] => http://www.w3school.com.cn
  )
)
```

PHP pathinfo() 函数

定义和用法

pathinfo() 函数以数组的形式返回文件路径的信息。

语法

```
pathinfo(path,options)
```

参数	描述
path	必需。规定要检查的路径。
process_sections	可选。规定要返回的数组元素。默认是 all。可能的值：`PATHINFO_DIRNAME` - 只返回 dirname `PATHINFO_BASENAME` - 只返回 basename `PATHINFO_EXTENSION` - 只返回 extension

说明

pathinfo() 返回一个关联数组包含有 *path* 的信息。

包括以下的数组元素：

- [dirname]
- [basename]
- [extension]

提示和注释

注释：如果不是要求取得所有单元，则 pathinfo() 函数返回字符串。

例子

例子 1

```
<?php
print_r(pathinfo("/testweb/test.txt"));
?>
```

输出：

```
Array
(
[dirname] => /testweb
[basename] => test.txt
[extension] => txt
)
```

例子 2

```
<?php
print_r(pathinfo("/testweb/test.txt",PATHINFO_BASENAME));
?>
```

输出：

```
test.txt
```

PHP pclose() 函数

定义和用法

pclose() 函数关闭由 popen() 打开的管道。

语法

```
pclose(pipe)
```

参数	描述
pipe	必需。规定由 popen() 打开的管道。

说明

该函数返回运行的进程的终止状态。

若出错，则返回 false。

例子

```
<?php
$file = popen("/bin/ls","r");

//一些要执行的代码

pclose($file);
?>
```

PHP popen() 函数

定义和用法

popen() 函数打开进程文件指针。

语法

```
popen(command,mode)
```

参数	描述
command	必需。规定要执行的命令。
mode	必需。规定连接模式。 可能的值： r：只读。 w：只写 (打开并清空已有文件或创建一个新文件)

说明

打开一个指向进程的管道，该进程由派生指定的 *command* 命令执行而产生。

返回一个和 fopen() 所返回的相同的文件指针，只不过它是单向的（只能用于读或写）并且必须用 pclose() 来关闭。此指针可以用于 fgets()，fgetss() 和 fwrite()。

若出错，则返回 false。

例子

```
<?php
$file = popen("/bin/ls","r");

//一些要执行的代码

pclose($file);
?>
```

PHP readfile() 函数

定义和用法

readfile() 函数输出一个文件。

该函数读入一个文件并写入到输出缓冲。

若成功，则返回从文件中读入的字节数。若失败，则返回 false。您可以通过 @readfile() 形式调用该函数，来隐藏错误信息。

语法

```
readfile(_filename_,_include_path_,_context_)
```

参数	描述
filename	必需。规定要读取的文件。
include_path	可选。如果也想在 *include_path* 中搜索文件，可以使用该参数并将其设为 true。
context	可选。规定文件句柄的环境。*Context* 是可以修改流的行为的一套选项。

说明

对 *context* 参数的支持是 PHP 5.0.0 添加的。

提示和注释

提示：如果在 php.ini 文件中 "fopen wrappers" 已经被激活，则在本函数中可以把 URL 作为文件名来使用。

例子

```
<?php
echo readfile("test.txt");
?>
```

输出：

```
There are two lines in this file.
This is the last line.
57
```

PHP readlink() 函数

定义和用法

readlink() 函数返回符号连接指向的目标。

若成功，则该函数返回连接的目标。若失败，则返回 false。

语法

```
readlink(linkpath)
```

参数	描述
linkpath	必需。规定要检查的连接路径。

提示和注释

注释：本函数未在 Windows 平台下实现。。

例子

```
<?php
echo readlink("/user/testlink");
?>
```

PHP realpath() 函数

定义和用法

realpath() 函数返回绝对路径。

该函数删除所有符号连接（比如 '/./', '/../' 以及多余的 '/'），返回绝对路径名。

若失败，则返回 false。比如说文件不存在的话。

语法

```
readlink(linkpath)
```

参数	描述
linkpath	必需。规定要检查的连接路径。

说明

在 BSD 系统上，如果仅仅是 *linkpath* 不存在的话，PHP 并不会像其它系统那样返回 false。

例子

```
<?php
echo realpath("test.txt");
?>
```

输出：

```
C:\Inetpub\testweb\test.txt
```

PHP rename() 函数

定义和用法

rename() 函数重命名文件或目录。

若成功，则该函数返回 true。若失败，则返回 false。

语法

```
rename(_oldname_,_newname_,_context_)
```

参数	描述
oldname	必需。规定要重命名的文件或目录。
newname	必需。规定文件或目录的新名称。
context	可选。规定文件句柄的环境。*context* 是可修改流的行为的一套选项。

提示和注释

注释：在 PHP 4.3.3 之前，rename() 不能在基于 *nix 的系统中跨磁盘分区重命名文件。

注释：用于 *oldname* 中的封装协议必须和用于 *newname* 中的相匹配。

注释：对 *context* 的支持是 PHP 5.0.0 添加的。

例子

```
<?php
rename("images","pictures");
?>
```

PHP rewind() 函数

定义和用法

rewind() 函数将文件指针的位置倒回文件的开头。

若成功，则返回 true。若失败，则返回 false。

语法

```
rewind(file)
```

参数	描述
file	必需。规定已打开的文件。

例子

```
<?php
$file = fopen("test.txt","r");

//改变文件指针的位置
fseek($file,"15");

//把文件指针设定为 0
rewind($file);

fclose($file);
?>
```

PHP rmdir() 函数

定义和用法

rmdir() 函数删除空的目录。

若成功，则该函数返回 true。若失败，则返回 false。

语法

```
rmdir(dir,context)
```

参数	描述
dir	必需。规定要删除的目录。
context	必需。规定文件句柄的环境。Context 是可修改流的行为的一套选项。

说明

尝试删除 *dir* 所指定的目录。 该目录必须是空的，而且要有相应的权限。

提示和注释

注释：对 *context* 的支持是 PHP 5.0.0 添加的。

例子

```
<?php
$path = "images";
if(!rmdir($path))
  {
  echo ("Could not remove $path");
  }
?>
```

PHP set_file_buffer() 函数

定义和用法

set_file_buffer() 函数设置打开文件的缓冲大小。

若成功，则该函数返回 0。若失败，则返回 EOF。

语法

```
set_file_buffer(file,buffer)
```

参数	描述
file	必需。规定打开的文件。
buffer	必需。规定缓冲大小，以字节计。

提示和注释

注释：该函数是 stream_set_write_buffer() 的别名。

例子

创建无缓冲的流：

```
<?php
$file = fopen("test.txt","w");
if ($file)
  {
  set_file_buffer($file,0);
  fwrite($file,"Hello World. Testing!");
  fclose($file);
  }
?>
```

PHP stat() 函数

定义和用法

stat() 函数返回关于文件的信息。

语法

```
fstat(file)
```

参数	描述
file	必需。规定要检查的文件。

说明

获取由 *file* 指定的文件的统计信息。如果 *file* 是符号连接，则统计信息是关于被连接文件本身的，而不是符号连接。

如果出错，stat() 返回 false，并且发出一条警告。

返回的数组包含有文件的统计信息，该数组具有以下列出的单元，数组下标从零开始。除了数字索引之外，从 PHP 4.0.6 起还可以通过关联索引来访问。

stat() 的返回格式

数字下标	关联键名（自 PHP 4.0.6）	说明
0	dev	设备名
1	ino	号码
2	mode	inode 保护模式
3	nlink	被连接数目
4	uid	所有者的用户 id
5	gid	所有者的组 id
6	rdev	设备类型，如果是 inode 设备的话
7	size	文件大小的字节数
8	atime	上次访问时间（Unix 时间戳）
9	mtime	上次修改时间（Unix 时间戳）
10	ctime	上次改变时间（Unix 时间戳）
11	blksize	文件系统 IO 的块大小
12	blocks	所占据块的数目

提示和注释

提示：lstat() 与 stat() 类似，不同的是，它会返回符号连接的状态。

注释：本函数的结果会被缓存。请使用 clearstatcache() 来清除缓存。

例子

```
<?php
$file = fopen("test.txt","r");
print_r(stat($file));
fclose($file);
?>
```

输出类似：

```
Array
(
[0] => 0
[1] => 0
[2] => 33206
[3] => 1
[4] => 0
[5] => 0
[6] => 0
[7] => 92
[8] => 1141633430
[9] => 1141298003
[10] => 1138609592
[11] => -1
[12] => -1
[dev] => 0
[ino] => 0
[mode] => 33206
[nlink] => 1
[uid] => 0
[gid] => 0
[rdev] => 0
[size] => 92
[atime] => 1141633430
[mtime] => 1141298003
[ctime] => 1138609592
[blksize] => -1
[blocks] => -1
)
```

PHP symlink() 函数

定义和用法

symlink() 函数创建符号连接。

语法

```
link(target,link)
```

参数	描述
target	必需。
link	必需。

说明

symlink() 对于已有的 *target* 建立一个名为 *link* 的符号连接。

若成功则返回 true，失败则返回 false。

提示和注释

注释：本函数未在 Windows 平台下实现。

PHP tempnam() 函数

定义和用法

tempnam() 函数创建一个具有唯一文件名的临时文件。

若成功，则该函数返回新的临时文件名。若失败，则返回 false。

语法

```
tempnam(dir,prefix)
```

参数	描述
dir	必需。规定创建临时文件的目录。
prefix	必需。规定文件名的开头。

说明

在指定目录中建立一个具有唯一文件名的文件。如果该目录不存在，tempnam() 会在系统临时目录中生成一个文件，并返回其文件名。

在 PHP 4.0.6 之前，tempnam() 函数的行为取决于系统。在 Windows 下 TMP 环境变量会越过 dir 参数，在 Linux 下 TMPDIR 环境变量优先，而在 SVR4 下总是使用 dir 参数，如果其指向的目录存在的话。

提示和注释

注释：如果 PHP 不能在指定的 *dir* 参数中创建文件，则退回到系统默认值。

注释：本函数的行为在 4.0.3 版中改变了。也会建立一个临时文件以避免竞争情形，即有可能会在产生出作为文件名的字符串与脚本真正建立该文件之间会在文件系统中存在同名文件。注意，如果不再需要该文件则要删除此文件，不会自动删除的。

提示：参见 tmpfile()

例子

```
<?php
echo tempnam("C:\inetpub\testweb","TMP0");
?>
```

输出：

```
C:\inetpub\testweb\TMP1.tmp
```

PHP tmpfile() 函数

定义和用法

tmpfile() 函数以读写（w+）模式建立一个具有唯一文件名的临时文件。

文件会在关闭后（用 fclose()）自动被删除，或当脚本结束后。

语法

```
tmpfile()
```

提示和注释

提示：参见 tempnam()。

例子

```
<?php
$temp = tmpfile();

fwrite($temp, "Testing, testing.");

//倒回文件的开头
rewind($temp);

//从文件中读取 1k
echo fread($temp,1024);

//删除文件
fclose($temp);
?>
```

输出：

```
Testing, testing.
```

PHP touch() 函数

定义和用法

touch() 函数设置指定文件的访问和修改时间。

语法

```
touch(filename,time,atime)
```

参数	描述
filename	必需。规定要接触的文件。
time	可选。设置时间。默认是当前系统时间。
atime	可选。设置访问时间。默认是当前系统时间。

说明

尝试将由 *filename* 给出的文件的访问和修改时间设定为指定的时间。如果没有设置可选参数 *time*，则使用当前系统时间。如果给出了第三个参数 *atime*，则指定文件的访问时间会被设为 atime 。

如果成功则返回 true，失败则返回 false。

提示和注释

注释：如果文件不存在，则会被创建。

例子

```
<?php
touch("test.txt");
?>
```

PHP umask() 函数

定义和用法

umask() 函数改变当前的 umask。

umask() 将 PHP 的 umask 设定为 mask & 0777 并返回原来的 umask。当 PHP 被作为服务器模块使用时，在每个请求结束后 umask 会被恢复。

无参数调用 umask() 会返回当前的 umask。

语法

```
umask(mask)
```

参数	描述
mask	必需。规定新的权限。默认是 0777。

提示和注释

注释：在多线程的服务器上尽量避免使用这个函数。创建文件后要改变其权限最好还是使用 chmod()。使用 umask() 会导致并发程序和服务器发生不可预知的情况，因为它们使用相同的 umask。

PHP unlink() 函数

定义和用法

unlink() 函数删除文件。

若成功，则返回 true，失败则返回 false。

语法

```
unlink(filename,context)
```

参数	描述
filename	必需。规定要删除的文件。
context	可选。规定文件句柄的环境。Context 是可修改流的行为的一套选项。

提示和注释

注释：对 *context* 的支持是 PHP 5.0.0 添加的。

例子

```
<?php
$file = "test.txt";
if (!unlink($file))
  {
  echo ("Error deleting $file");
  }
else
  {
  echo ("Deleted $file");
  }
?>
```

PHP Filter 函数

PHP Filter 简介

PHP 过滤器用于对来自非安全来源的数据（比如用户输入）进行验证和过滤。

安装

filter 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP Filter 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
filter_has_var()	检查是否存在指定输入类型的变量。	5
filter_id()	返回指定过滤器的 ID 号。	5
filter_input()	从脚本外部获取输入，并进行过滤。	5
filter_input_array()	从脚本外部获取多项输入，并进行过滤。	5
filter_list()	返回包含所有得到支持的过滤器的一个数组。	5
filter_var_array()	获取多项变量，并进行过滤。	5
filter_var()	获取一个变量，并进行过滤。	5

PHP Filters

ID 名称	描述
FILTER_CALLBACK	调用用户自定义函数来过滤数据。
FILTER_SANITIZE_STRING	去除标签，去除或编码特殊字符。
FILTER_SANITIZE_STRIPPED	"string" 过滤器的别名。
FILTER_SANITIZE_ENCODED	URL-encode 字符串，去除或编码特殊字符。
	HTML 转义字符 '"<>& 以

FILTER_SANITIZE_SPECIAL_CHARS	及 ASCII 值小于 32 的字符。	
FILTER_SANITIZE_EMAIL	删除所有字符，除了字母、数字以及 !#$%&'*+-/=?^_`{	}~@.[]
FILTER_SANITIZE_URL	删除所有字符，除了字母、数字以及 $-_.+!*'(),{}	\^~[]`<>#%";/?:@&=
FILTER_SANITIZE_NUMBER_INT	删除所有字符，除了数字和 +-	
FILTER_SANITIZE_NUMBER_FLOAT	删除所有字符，除了数字、+- 以及 .,eE。	
FILTER_SANITIZE_MAGIC_QUOTES	应用 addslashes()。	
FILTER_UNSAFE_RAW	不进行任何过滤，去除或编码特殊字符。	
FILTER_VALIDATE_INT	在指定的范围以整数验证值。	
FILTER_VALIDATE_BOOLEAN	如果是 "1", "true", "on" 以及 "yes"，则返回 true，如果是 "0", "false", "off", "no" 以及 ""，则返回 false。否则返回 NULL。	
FILTER_VALIDATE_FLOAT	以浮点数验证值。	
FILTER_VALIDATE_REGEXP	根据 regexp，兼容 Perl 的正则表达式来验证值。	
FILTER_VALIDATE_URL	把值作为 URL 来验证。	
FILTER_VALIDATE_EMAIL	把值作为 e-mail 来验证。	
FILTER_VALIDATE_IP	把值作为 IP 地址来验证。	

PHP filter_has_var() 函数

定义和用法

filter_has_var() 函数检查是否存在指定输入类型的变量。

若成功，则返回 true，否则返回 false。

语法

```
filter_has_var(type, variable)
```

参数	描述
type	必需。规定要检查的类型。可能的值：`INPUT_GET` `INPUT_POST` `INPUT_COOKIE` `INPUT_SERVER` `INPUT_ENV`
variable	必需。规定要检查的变量。

例子

在本例中，输入变量 "name" 被发送到 PHP 页面：

```
<?php
if(!filter_has_var(INPUT_GET, "name"))
 {
 echo("Input type does not exist");
 }
else
 {
 echo("Input type exists");
 }
?>
```

输出类似：

```
Input type exists
```

PHP filter_id() 函数

定义和用法

filter_id() 函数返回指定过滤器的 ID 号。

若成功，则返回过滤器的 ID 号。如果该过滤器不存在，则返回 NULL。

语法

```
filter_id(filter_name)
```

参数	描述
type	必需。规定被获取 ID 号的过滤器。必须是过滤器名称（不是过滤器 ID 名）。请使用 filter_list() 函数来获取所有被支持的过滤器的名称。

例子

```
<?php
echo(filter_id("validate_email"));
?>
```

输出类似：

```
274
```

PHP filter_input() 函数

定义和用法

filter_input() 函数从脚本外部获取输入，并进行过滤。

本函数用于对来自非安全来源的变量进行验证，比如用户的输入。

本函数可从各种来源获取输入：

- INPUT_GET
- INPUT_POST
- INPUT_COOKIE
- INPUT_ENV
- INPUT_SERVER
- INPUT_SESSION (Not yet implemented)
- INPUT_REQUEST (Not yet implemented)

如果成功，则返回被过滤的数据，如果失败，则返回 false，如果 *variable* 参数未设置，则返回 NULL。

语法

```
filter_input(input_type, variable, filter, options)
```

参数	描述
input_type	必需。规定输入类型。参见上面的列表中可能的类型。
variable	规定要过滤的变量。
filter	可选。规定要使用的过滤器的 ID。默认是 FILTER_SANITIZE_STRING。请参见完整的 PHP Filter 函数参考手册，获得可能的过滤器。过滤器 ID 可以是 ID 名称 （比如 FILTER_VALIDATE_EMAIL），或 ID 号（比如 274）。
options	规定包含标志/选项的数组。检查每个过滤器可能的标志和选项。

例子

在本例中，我们使用 filter_input() 函数来过滤一个 POST 变量。所接受的 POST 变量是合法的 e-mail 地址。

```
<?php
if (!filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL))
 {
 echo "E-Mail is not valid";
 }
else
 {
 echo "E-Mail is valid";
 }
?>
```

输出类似：

```
E-Mail is valid
```

PHP filter_input_array() 函数

定义和用法

filter_input_array() 函数从脚本外部获取多项输入，并进行过滤。

本函数无需重复调用 filter_input()，对过滤多个输入变量很有用。

本函数可从各种来源获取输入：

- INPUT_GET
- INPUT_POST
- INPUT_COOKIE
- INPUT_ENV
- INPUT_SERVER
- INPUT_SESSION (Not yet implemented)
- INPUT_REQUEST (Not yet implemented)

如果成功，则返回被过滤的数据，如果失败，则返回 false。

语法

```
filter_input(input_type, args)
```

参数	描述
input_type	必需。规定输入类型。参见上面的列表中可能的类型。
args	可选。规定过滤器参数数组。合法的数组键是变量名。合法的值是过滤器 ID，或者规定过滤器、标志以及选项的数组。该参数也可以是一个单独的过滤器 ID，如果是这样，输入数组中的所有值由指定过滤器进行过滤。

提示和注释

提示：参见完整的 PHP Filter 参考手册，查看可与该函数一同使用的过滤器。

例子

在本例中，我们使用 filter_input_array() 函数来过滤三个 POST 变量。所接受的 POST 变量是姓名、年龄以及电子邮件地址：

```
<?php
$filters = array
 (
 "name" => array
  (
  "filter"=>FILTER_CALLBACK,
  "flags"=>FILTER_FORCE_ARRAY,
  "options"=>"ucwords"
  ),
 "age" => array
  (
  "filter"=>FILTER_VALIDATE_INT,
  "options"=>array
   (
   "min_range"=>1,
   "max_range"=>120
   )
  ),
 "email"=> FILTER_VALIDATE_EMAIL,
 );
print_r(filter_input_array(INPUT_POST, $filters));
?>
```

输出类似：

```
Array
 (
 [name] => Peter
 [age] => 41
 [email] => peter@example.com
 )
```

PHP filter_list() 函数

定义和用法

filter_list() 函数返回包含所有得到支持的过滤器的一个数组。

语法

```
filter_list()
```

提示和注释

注释：该函数的结果不是过滤器的 ID，而是过滤器名称。请使用 filter_id() 函数来获取过滤器 ID。

例子

```
<?php
print_r(filter_list());
?>
```

输出类似：

```
Array
 (
 [0] => int
 [1] => boolean
 [2] => float
 [3] => validate_regexp
 [4] => validate_url
 [5] => validate_email
 [6] => validate_ip
 [7] => string
 [8] => stripped
 [9] => encoded
 [10] => special_chars
 [11] => unsafe_raw
 [12] => email
 [13] => url
 [14] => number_int
 [15] => number_float
 [16] => magic_quotes
 [17] => callback
 )
```

PHP filter_var_array() 函数

定义和用法

filter_var_array() 函数获取多项变量，并进行过滤。

由于无需重复调用 filter_input()，因此本函数对过滤多个变量很有用。

如果成功，则返回包含被过滤的变量值的数组，如果失败，则返回 false。

语法

```
filter_var_array(array, args)
```

参数	描述
array	必需。规定带有字符串键的数组，包含要过滤的数据。
args	可选。规定过滤器参数数组。合法的数组键是变量名。合法的值是过滤器 ID，或者规定过滤器、标志以及选项的数组。该参数也可以是一个单独的过滤器 ID，如果是这样，输入数组中的所有值由指定过滤器进行过滤。

提示和注释

提示：参见完整的 PHP Filter 参考手册，查看可与该函数一同使用的过滤器。

例子

```
<?php
$arr = array
 (
 "name" => "peter griffin",
 "age" => "41",
 "email" => "peter@example.com",
 );

$filters = array
 (
 "name" => array
  (
  "filter"=>FILTER_CALLBACK,
  "flags"=>FILTER_FORCE_ARRAY,
  "options"=>"ucwords"
  ),
 "age" => array
  (
  "filter"=>FILTER_VALIDATE_INT,
  "options"=>array
   (
   "min_range"=>1,
   "max_range"=>120
   )
  ),
 "email"=> FILTER_VALIDATE_EMAIL,
 );

print_r(filter_var_array($arr, $filters));
?>
```

输出类似：

```
Array
 (
 [name] => Peter Griffin
 [age] => 41
 [email] => peter@example.com
 )
```

PHP filter_var() 函数

定义和用法

filter_var() 函数通过指定的过滤器过滤变量。

如果成功，则返回已过滤的数据，如果失败，则返回 false。

语法

```
filter_var(variable, filter, options)
```

参数	描述
variable	必需。规定要过滤的变量。
filter	可选。规定要使用的过滤器的 ID。
options	规定包含标志/选项的数组。检查每个过滤器可能的标志和选项。

提示和注释

提示：参见完整的 PHP Filter 参考手册，查看可与该函数一同使用的过滤器。

例子

```
<?php
if(!filter_var("someone@example....com", FILTER_VALIDATE_EMAIL))
 {
 echo("E-mail is not valid");
 }
else
 {
 echo("E-mail is valid");
 }
?>
```

输出类似：

```
E-mail is not valid
```

PHP FTP 函数

PHP FTP 简介

FTP 函数通过文件传输协议 (FTP) 提供对文件服务器的客户端访问。

FTP 函数用于打开、登录以及关闭连接，同时用于上传、下载、重名命、删除及获取文件服务器上的文件信息。不是所有 FTP 函数对每个服务器都起作用或返回相同的结果。自 PHP 3 起，FTP 函数可用。

这些函数用于对 FTP 服务器进行细致的访问。如果您仅仅需要对 FTP 服务器进行读写操作，建议使用 Filesystem 函数中的 ftp:// wrapper。

安装

PHP 的 Windows 版本已经内置该 FTP 扩展模块的支持。无需加载任何附加扩展库即可使用这些函数。

不过，如果您运行的是 PHP 的 Linux 版本，在编译的时候请添加 --enable-ftp 选项 (PHP4 或以上版本) 或者 --with-ftp (PHP3 版本)。

PHP FTP 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
ftp_alloc()	为要上传到 FTP 服务器的文件分配空间。	5
ftp_cdup()	把当前目录改变为 FTP 服务器上的父目录。	3
ftp_chdir()	改变 FTP 服务器上的当前目录。	3
ftp_chmod()	通过 FTP 设置文件上的权限。	5
ftp_close()	关闭 FTP 连接。	4
ftp_connect()	打开 FTP 连接。	3
ftp_delete()	删除 FTP 服务器上的文件。	3
ftp_exec()	在 FTP 上执行一个程序/命令。	4
ftp_fget()	从 FTP 服务器上下载一个文件并保存到本地一个已经打开的文件中。	3

ftp_fput()	上传一个已打开的文件，并在 FTP 服务器上把它保存为一个文件。	3
ftp_get_option()	返回当前 FTP 连接的各种不同的选项设置。	4
ftp_get()	从 FTP 服务器下载文件。	3
ftp_login()	登录 FTP 服务器。	3
ftp_mdtm()	返回指定文件的最后修改时间。	3
ftp_mkdir()	在 FTP 服务器创建一个新目录。	3
ftp_nb_continue()	连续获取／发送文件 (non-blocking)。	4
ftp_nb_fget()	从FTP服务器上下载文件并保存到本地已经打开的文件中 (non-blocking)	4
ftp_nb_fput()	上传已打开的文件，并在FTP服务器上把它保存为文件 (non-blocking)。	4
ftp_nb_get()	从 FTP 服务器下载文件 (non-blocking)。	4
ftp_nb_put()	把文件上传到服务器 (non-blocking)。	4
ftp_nlist()	返回指定目录的文件列表。	3
ftp_pasv()	返回当前 FTP 被动模式是否打开。	3
ftp_put()	把文件上传到服务器。	3
ftp_pwd()	返回当前目录名称。	3
ftp_quit()	ftp_close() 的别名。	3
ftp_raw()	向 FTP 服务器发送一个 raw 命令。	5
ftp_rawlist()	返回指定目录中文件的详细列表。	3
ftp_rename()	重命名 FTP 服务器上的文件或目录。	3
ftp_rmdir()	删除 FTP 服务器上的目录。	3
ftp_set_option()	设置各种 FTP 运行时选项。	4
ftp_site()	向服务器发送 SITE 命令。	3
ftp_size()	返回指定文件的大小。	3
ftp_ssl_connect()	打开一个安全的 SSL-FTP 连接。	4
ftp_systype()	返回远程 FTP 服务器的系统类型标识符。	3

PHP FTP 常量

PHP：指示支持该常量的最早的 PHP 版本。

常量	描述	PHP
FTP_ASCII		3
FTP_TEXT		3
FTP_BINARY		3
FTP_IMAGE		3
FTP_TIMEOUT_SEC		3
FTP_AUTOSEEK		4
FTP_AUTORESUME	为 GET 和 PUT 请求自动决定恢复和开始的位置 只能工作在 FTP_AUTOSEEK 打开的情况下	4
FTP_FAILED	异步传输失败	4
FTP_FINISHED	异步传输成功	4
FTP_MOREDATA	异步传输是活动状态的	4

PHP ftp_alloc() 函数

定义和用法

ftp_alloc() 函数为要上传到 FTP 服务器的文件分配空间。

若成功，则返回 true。否则返回 false。

语法

```
ftp_alloc(ftp_connection,size,return)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接。
size	可选。规定要分配的字节数。
return	可选。规定存储服务器响应的字节数。

提示和注释

注释：很多服务器不支持该命令。

例子

例子 1

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_alloc($conn,"160",$response);
echo $response;

ftp_close($conn);
?>
```

例子 2

```
<?php

$file = "myfile.txt";

$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

if (ftp_alloc($conn, filesize($file), $response))
  {
  echo "Space allocated on server.";
  }
else
  {
  echo "Unable to allocate space. " . $response;
  }

ftp_close($conn);

?>
```

PHP ftp_cdup() 函数

定义和用法

ftp_cdup() 函数把当前目录改变为 FTP 服务器上的父目录。

若成功，则返回 true。否则返回 false。

语法

```
ftp_cdup(ftp_connection)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

//输出当前目录
echo "Dir: ".ftp_pwd($conn);
echo "<br />";

//更改为 images 目录
ftp_chdir($conn,"images");
echo "Dir: ".ftp_pwd($conn);
echo "<br />";

//把当前目录切换为父目录
ftp_cdup($conn);
echo "Dir: ".ftp_pwd($conn);

ftp_close($ftp_server);
?>
```

输出：

```
Dir: /
Dir: /images
Dir: /
```

PHP ftp_chdir() 函数

定义和用法

ftp_chdir() 函数改变 FTP 服务器上的当前目录。

若成功，则返回 true。否则返回 false。如果切换目录失败，PHP 还会发出一条警告。

语法

```
ftp_chdir(ftp_connection,directory)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
directory	必需。规定要切换到的目录。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

//输出当前目录
echo "Dir: ".ftp_pwd($conn);
echo "<br />";

//切换为 images 目录
ftp_chdir($conn,"images");
echo "Dir: ".ftp_pwd($conn);

ftp_close($ftp_server);
?>
```

输出：

```
Dir: /
Dir: /images
```

PHP ftp_chmod() 函数

定义和用法

ftp_chmod() 函数设置 FTP 服务器上指定文件的权限。

若成功，则该函数返回新的权限。否则返回 false。

语法

```
ftp_chmod(ftp_connection,mode,file)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
mode	必需。规定新的权限。
file	必需。规定要修改权限的文件的名称。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"user","pass");

// 所有者可读写，其他人没有任何权限
ftp_chmod($conn,"0600","test.txt");

// 所有者可读写，其他人可读
ftp_chmod($conn,"0644","test.txt");

// 所有者拥有所有权限，其他人可读可执行
ftp_chmod($conn,"0755","test.txt");

// 所有者拥有所有权限，所有者所属的组可读
ftp_chmod($conn,"0740","test.txt");

ftp_close($conn);
?>
```

PHP ftp_close() 函数

定义和用法

ftp_close() 函数关闭 FTP 连接。

该函数关闭给出的连接标识符并释放资源。

语法

```
ftp_close(ftp_connection)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");

//要执行的一些代码

ftp_close($conn);
?>
```

PHP ftp_connect() 函数

定义和用法

ftp_connect() 函数建立一个新的 FTP 连接。

若成功，则返回一个连接标识，否则返回 false。

语法

```
ftp_connect(host,port,timeout)
```

参数	描述
host	必需。规定要连接的 FTP 服务器。可以是域名或 IP 地址。后面不应以斜线结尾，前面也不需要用 ftp:// 开头。
port	可选。规定 FTP 服务器的端口。
timeout	可选。规定该 FTP 服务器的超时时间。默认是 90 秒。

说明

提示：超时时间可以在任何时候通过函数 ftp_set_option() 及 ftp_get_option() 来改变和获取。

参数 *timeout* 仅适用于 PHP 4.2.0 及以上版本。

例子

本例尝试连接一个 FTP 服务器。如果连接失败，则 die() 函数将终止脚本，并输出一条消息：

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
?>
```

PHP ftp_delete() 函数

定义和用法

ftp_delete() 函数删除 FTP 服务器上的一个文件。

若成功，则返回 true，否则返回 false。

语法

```
ftp_delete(ftp_connection,path)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
path	必需。规定要删除的文件的路径。

说明

ftp*delete()* 函数用来删除 *FTP* 服务器上的一个由参数 *_path* 指定的的文件。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_delete($conn,"test.txt");

ftp_close($conn);
?>
```

输出：

```
1
```

PHP ftp_exec() 函数

定义和用法

ftp_exec() 函数请求在 FTP 服务器上执行一个程序或命令。

若成功（服务器发送响应代码 200），则返回 true，否则返回 false。

语法

```
ftp_exec(ftp_connection,command)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
command	必需。规定发送到服务器的命名请求。

说明

该函数发送一个 SITE EXEC command 请求到 FTP 服务器。

提示和注释

与 ftp_raw() 函数 不同，ftp_exec() 只有在登录到 FTP 服务器后才能发送命令。

例子

```
<?php
$command = "ls-al > test.txt";
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

if (ftp_exec($conn,$command))
  {
  echo "Command executed successfully";
  }
else
  {
  echo "Execution of command failed";
  }

ftp_close($conn);
?>
```

PHP ftp_fget() 函数

定义和用法

ftp_fget() 函数从 FTP 服务器上下载一个文件并保存到本地一个已经打开的文件中。

若成功则返回 true，失败则返回 false。

语法

```
ftp_fget(ftp_connection,local,remote,mode,resume)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
local	必需。本地已经打开的文件的句柄。
remote	必需。规定从中进行拷贝的文件的路径。
mode	必需。规定传输模式。可能的值有： `FTP_ASCII` `FTP_BINARY`
resume	必需。规定在远程文件中的何处开始拷贝。默认是 0。

说明

参数 *resume* 仅适用于 PHP 4.3.0 以上版本

例子

本例把文本从 "source.txt" 拷贝到 "target.txt" 中：

```
<?php
$source = "source.txt";
$target = fopen("target.txt", "w");

$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_fget($conn,$target,$source,FTP_ASCII);

ftp_close($conn);
?>
```

PHP ftp_fput() 函数

定义和用法

ftp_fput() 函数上传一个已经打开的文件到 FTP 服务器。

若成功则返回 true，失败则返回 false。

语法

```
ftp_fput(ftp_connection,remote,local,mode,resume)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
remote	必需。上传到服务器上的文件名。
local	必需。规定所打开文件的句柄。
mode	必需。规定传输模式。可能的值有： `FTP_ASCII` `FTP_BINARY`
resume	必需。规定在本地文件中的何处开始拷贝。默认是 0。

说明

参数 *resume* 仅适用于 PHP 4.3.0 以上版本

例子

本例把文本从 "source.txt" 拷贝到 "target.txt" 中：

```
<?php
$source = fopen("source.txt","r");

$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_fput($conn,"target.txt",$source,FTP_ASCII);

ftp_close($conn);
?>
```

输出：

```
1
```

PHP ftp_get_option() 函数

定义和用法

ftp_get_option() 函数返回当前 FTP 连接的各种不同的选项设置。

语法

```
ftp_get_option(ftp_connection,option)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
option	必需。规定要返回的选项。可能的值有： `FTP_TIMEOUT_SEC` - 返回网络操作的时间限制 `FTP_AUTOSEEK` - 如果设置该选项，则返回 true，否则返回 false

说明

如果成功，则返回选项的值，否则，如果给定的参数 *option* 选项若不被支持，则返回 false 同时会提示一条错误信息。

此函数会返回连接句柄为 *ftp_connection*，指定键值 *option* 的值。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_get_option($conn,FTP_TIMEOUT_SEC);

ftp_close($conn);
?>
```

输出：

```
90
```

PHP ftp_get() 函数

定义和用法

ftp_get() 函数从 FTP 服务器上下载一个文件。

若成功则返回 true，失败则返回 false。

语法

```
ftp_get(ftp_connection,local,remote,mode,resume)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
local	必需。规定本地文件。
remote	必需。规定从中进行拷贝的文件的路径。
mode	必需。规定传输模式。可能的值有： FTP_ASCII FTP_BINARY
resume	必需。规定在远程文件中的何处开始拷贝。默认是 0。

说明

参数 *resume* 仅适用于 PHP 4.3.0 以上版本

例子

本例把文本从 "source.txt" 拷贝到 "target.txt" 中：

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_get($conn,"target.txt","source.txt",FTP_ASCII);

ftp_close($conn);
?>
```

输出：

```
1
```

PHP ftp_login() 函数

定义和用法

ftp_login() 函数登录 FTP 服务器。

若成功则返回 true，失败则返回 false 并发出一个警告。

语法

```
ftp_login(ftp_connection,username,password)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
username	必需。规定用于登录的用户名。
password	必需。规定用于登录的密码。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");
ftp_close($conn);
?>
```

PHP ftp_mdtm() 函数

定义和用法

ftp_mdtm() 函数返回指定文件的最后修改时间。

语法

```
ftp_login(ftp_connection,file)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
file	必需。规定要检查的文件。

提示和注释

注释：并非所有 FTP 服务器都支持该函数。

注释：该函数不适用于检查目录。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

$mod = ftp_mdtm($conn,"test.txt");

//以 Unix 时间戳返回结果
echo $mod;
echo "<br />";

//把 Unix 时间戳格式化为日期
echo date(DATE_RFC822,$mod);

ftp_close($conn);
?>
```

输出：

```
1140082571
Thu, 16 Feb 2006 10:36:11 CET
```

PHP ftp_mkdir() 函数

定义和用法

ftp_mkdir() 函数在 FTP 服务器上建立新目录。

语法

```
ftp_mkdir(ftp_connection,dir)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
dir	必需。规定要创建的目录的名称。

说明

在 FTP 服务器上建立一个目录名为参数 *dir* 的新目录。

如果成功，则返回新建的目录名，否则返回 false。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_mkdir($conn,"testdir");

ftp_close($conn);
?>
```

输出：

```
/testdir
```

PHP ftp_nb_continue() 函数

定义和用法

ftp_nb_continue() 函数连续获取 / 发送文件。

该函数返回下列值：

- FTP_FAILED (send/receive failed)
- FTP_FINISHED (send/receive completed)
- FTP_MOREDATA (send/receive in progress)

该函数异步地发送/获取文件。这意味着您的程序可以在文件下载时执行其他操作。

语法

```
ftp_nb_continue(ftp_connection)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。

例子

```
<?php
$source = "source.txt";
$target = fopen("target.txt", "w");

$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

$status = ftp_nb_fget($conn,$source,$target,FTP_ASCII);

while ($status == FTP_MOREDATA)
  {
  $status = ftp_nb_continue($conn);
  }

if ($status != FTP_FINISHED)
  {
  echo "Download error";
  }

ftp_close($conn);
?>
```

PHP ftp_nb_fget() 函数

定义和用法

ftp_nb_fget() 函数从 FTP 服务器上下载一个文件并保存到本地已经打开的一个文件中 (non-blocking)。

该函数返回下列值：

- FTP_FAILED (send/receive failed)
- FTP_FINISHED (send/receive completed)
- FTP_MOREDATA (send/receive in progress)

与 ftp_fget() 不同，该函数异步地获取文件。这意味着您的程序可以在文件下载时执行其他操作。

语法

```
ftp_nb_fget(ftp_connection,local,remote,mode,resume)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
local	必需。规定本地文件。
remote	必需。规定从中进行拷贝的文件的路径。
mode	必需。规定传输模式。可能的值有： `FTP_ASCII` `FTP_BINARY`
resume	必需。规定在远程文件中的何处开始拷贝。默认是 0。

例子

本例把文本从 "source.txt" 拷贝到 "target.txt" 中：

```
<?php
$source = "source.txt";
$target = fopen("target.txt", "w");

$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_nb_fget($conn,$target,$source,FTP_ASCII);

ftp_close($conn);
?>
```

PHP ftp_nb_put() 函数

定义和用法

ftp_nb_put() 函数把文件上传到服务器 (non-blocking)。

该函数返回下列值：

- FTP_FAILED (send/receive failed)
- FTP_FINISHED (send/receive completed)
- FTP_MOREDATA (send/receive in progress)

与 ftp_put() 不同，该函数异步地获取文件。这意味着您的程序可以在文件传输时执行其他操作。

语法

```
ftp_nb_fput(ftp_connection,remote,local,mode,resume)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
remote	必需。上传到服务器上的文件名。
local	必需。规定要上传的本地文件的路径。
mode	必需。规定传输模式。可能的值有： FTP_ASCII FTP_BINARY
resume	必需。规定在本地文件中的何处开始拷贝。默认是 0。

例子

本例把文本从 "source.txt" 拷贝到 "target.txt" 中：

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_nb_put($conn,"target.txt","source.txt",FTP_ASCII);

ftp_close($conn);
?>
```

PHP ftp_nlist() 函数

定义和用法

ftp_nlist() 函数返回指定目录的文件列表。

如果成功，则返回给定目录下的文件名组成的数组，否则返回 false。

语法

```
ftp_nlist(ftp_connection,dir)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
dir	必需。规定要检查的目录。使用 "." 来获得当前目录。

提示和注释

注释：该函数不适用于 IIS (Internet Information Server)。它返回 nothing。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

print_r(ftp_nlist($conn,"images"));

ftp_close($conn);
?>
```

输出类似：

```
array(3)
{
[0]=> "flower.gif"
[1]=> "car.gif"
[2]=> "house.gif"
}
```

PHP ftp_pasv() 函数

定义和用法

ftp_pasv() 函数把被动模式设置为打开或关闭。

在被动模式中，数据连接是由客户机来初始化的，而不是服务器。这在客户机位于防火墙之后时比较有用。

语法

```
ftp_pasv(ftp_connection,mode)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
mode	必需。规定模式。 TRUE = passive mode on FALSE = passive mode off

说明

如果参数 *mode* 为真，打开被动模式传输 (PASV MODE)，否则，如果参数 *mode* 为假，则关闭被动传输模式。在被动模式打开的情况下，数据的传送由客户机启动，而不是由服务器开始。

如果成功则返回 true，失败则返回 false。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_pasv($conn,TRUE);

ftp_close($conn);
?>
```

PHP ftp_put() 函数

定义和用法

ftp_put() 函数把文件上传到服务器。

若成功则返回 true，失败则返回 false。

语法

```
ftp_put(ftp_connection,remote,local,mode,resume)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
remote	必需。上传到服务器上的文件名。
local	必需。规定要上传的本地文件的路径。
mode	必需。规定传输模式。可能的值有： FTP_ASCII FTP_BINARY
resume	必需。规定在本地文件中的何处开始拷贝。默认是 0。

例子

本例把文本从 "source.txt" 拷贝到 "target.txt" 中：

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_put($conn,"target.txt","source.txt",FTP_ASCII);

ftp_close($conn);
?>
```

输出：

```
1
```

PHP ftp_pwd() 函数

定义和用法

ftp_pwd() 函数返回当前目录名。

语法

```
ftp_pwd(ftp_connection)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

// 输出当前目录
echo ftp_pwd($conn) . "<br />";

// 把目录改为 images
ftp_chdir($conn,"images");

// 输出当前目录
echo ftp_pwd($conn);

ftp_close($conn);
?>
```

PHP ftp_quit() 函数

定义和用法

ftp_quit() 函数关闭 FTP 连接。

该函数关闭给出的连接标识符并释放资源。

语法

```
ftp_quit(ftp_connection)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。

提示和注释

提示：该函数是 ftp_close() 函数的别名。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");

//要执行的一些代码

ftp_quit($conn);
?>
```

PHP ftp_raw() 函数

定义和用法

ftp_raw() 函数向 FTP 服务器发送一个 raw 命令。

语法

```
ftp_raw(ftp_connection,command)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
command	必需。规定要执行的命令。

提示和注释

注释：该函数以字符串数组的形式返回服务器的响应。不执行解析，且 ftp_raw() 不检查命令是否正确。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");

print_r (ftp_raw($conn,"USER admin"));
print_r (ftp_raw($conn,"PASS ert456"));

ftp_close($conn);
?>
```

输出：

```
Array ([0] => 331 User admin, password please)
Array ([0] => 230 Password Ok, User logged in)
```

PHP ftp_rawlist() 函数

定义和用法

ftp_rawlist() 函数返回指定目录中文件的详细列表。

语法

```
ftp_rawlist(ftp_connection,dir,recursive)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
dir	必需。规定目录。使用 "." 来规定当前目录。
recursive	可选。默认地，该函数向服务器发送 "LIST" 命令。如果，如果 recursive 参数设置为 true，则发送 "LIST -R" 命令。

说明

ftp_rawlist() 函数将执行 FTP LIST 命令，并把结果返回为一个数组。数组的每个元素为返回文本的每一行，输出结构不会被解析。

使用函数 ftp_systype() 可以用来判断 FTP 服务器的类型，从而可以用来判断返回列表的类型。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

print_r (ftp_rawlist($conn,"."));

ftp_close($conn);
?>
```

输出类似：

```
Array
(
[0] => dr--r--r-- 1 user group 0 Feb 15 13:02 .
[1] => dr--r--r-- 1 user group 0 Feb 15 13:02 ..
[2] => drw-rw-rw- 1 user group 0 Jan 03 08:33 images
[3] => -rw-rw-rw- 1 user group 160 Feb 16 13:54 test.php
[4] => -rw-rw-rw- 1 user group 20 Feb 14 12:22 test.txt
)
```

PHP ftp_rename() 函数

定义和用法

ftp_rename() 函数更改 FTP 服务器上的文件或目录名。

如果成功，则返回 true，否则返回 false。

语法

```
ftp_rename(ftp_connection,from,to)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
from	必需。规定要改名的文件或目录。
to	必需。规定文件或目录的新名称。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_rename($conn,"oldname.txt","newname.txt");

ftp_close($conn);
?>
```

PHP ftp_rmdir() 函数

定义和用法

ftp_rmdir() 函数删除一个目录。

如果成功，则返回 true，否则返回 false。

语法

```
ftp_rmdir(ftp_connection,dir)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
dir	必需。规定要删除的目录。

说明

删除由参数 *dir* 指定的目录。*dir* 必须是一个空目录的绝对或相对路径。

如果成功，则返回 true，否则返回 false。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_rmdir($conn,"testdir");

ftp_close($conn);
?>
```

PHP ftp_set_option() 函数

定义和用法

ftp_set_option() 函数设置各种 FTP 运行时选项。

语法

```
ftp_set_option(ftp_connection,option,value)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
option	必需。规定要删设置的运行时选项。可能的值： FTP_TIMEOUT_SEC FTP_AUTOSEEK 详细信息见下面的说明。
value	必需。设置 option 参数的值。

说明

FTP_TIMEOUT_SEC 选项改变网络传输的超时时间。参数 value 必须为整数且大于 0。默认的超时时间为 90 秒。

当 FTP_AUTOSEEK 选项打开时，带 resumepos 或 startpos 参数的 GET 或 PUT 请求将先检索到文件中指定的位置。此选项默认是打开的。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_set_option($conn,FTP_TIMEOUT_SEC,120);

ftp_close($conn);
?>
```

PHP ftp_site() 函数

定义和用法

ftp_site() 函数向服务器发送 SITE 命令。

SITE 命令没有标准化。不同的服务器不尽相同。对于处理文件权限或组关系方面的事情，SITE 命令很有用。

语法

```
ftp_site(ftp_connection,command)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
command	必需。规定向 FTP 发送的 SITE 命令。

说明

ftp*site() 函数向 FTP 服务器发送由参数 _command* 指定的命令。

如果成功则返回 true，失败则返回 false。

提示和注释

提示：如需查看哪些命令可用，请通过 ftp_raw() 函数发送 REMOTEHELP 命令。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

ftp_site($conn,"CHMOD 0600 sitetest.txt");

ftp_close($conn);
?>
```

PHP ftp_size() 函数

定义和用法

ftp_size() 函数返回指定文件的大小。

语法

```
ftp_size(ftp_connection,remote_file)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。
remote_file	必需。规定要检查的文件。

说明

ftp*size()* 函数以字节返回远程文件 _*remote_file* 的大小。如果指定文件不存在或发生错误，则返回 -1。有些 FTP 服务器可能不支持此特性。

如果获取成功，则返回文件大小，否则返回 -1。

提示和注释

注释：并非所有 FTP 服务器均支持该函数。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_size($conn,"test.txt");

ftp_close($conn);
?>
```

输出类似：

```
160
```

PHP ftp_ssl_connect() 函数

定义和用法

ftp_ssl_connect() 函数打开一个安全的 SSL-FTP 连接。

当连接打开，您就可以在服务器运行 FTP 函数。

语法

```
ftp_ssl_connect(host,port,timeout)
```

参数	描述
host	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。可以是域名或 IP 地址。该参数不能包含 "ftp://" 或 斜杠。
port	可选。规定 FTP 服务器的端口。默认是 21。
timeout	可选。规定 FTP 连接的超时时间。默认是 90 秒。

例子

本例尝试连接一个 FTP 服务器。如果连接失败，die() 函数会终止脚本的执行，并输出一条消息：

```
<?php
$conn = ftp_ssl_connect("ftp.testftp.com") or die("Could not connect");
?>
```

PHP ftp_systype() 函数

定义和用法

ftp_systype() 函数返回远程 FTP 服务器的系统类型标识符。

该函数返回远程服务器的系统类型。若发生错误，则返回 false。

语法

```
ftp_systype(ftp_connection)
```

参数	描述
ftp_connection	必需。规定要使用的 FTP 连接（FTP 连接的标识符）。

例子

```
<?php
$conn = ftp_connect("ftp.testftp.com") or die("Could not connect");
ftp_login($conn,"admin","ert456");

echo ftp_systype($conn);

ftp_close($conn);
?>
```

输出类似：

```
UNIX
```

PHP HTTP 函数

PHP HTTP 简介

HTTP 函数允许您在其他输出被发送之前，对由 web 服务器发送到浏览器的信息进行操作。

安装

HTTP 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP HTTP 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
header()	向客户端发送原始的 HTTP 报头。	3
headers_list()	返回已发送的（或待发送的）响应头部的一个列表。	5
headers_sent()	检查 HTTP 报头是否发送/已发送到何处。	3
setcookie()	向客户端发送一个 HTTP cookie。	3
setrawcookie()	不对 cookie 值进行 URL 编码，发送一个 HTTP cookie。	5

PHP header() 函数

定义和用法

header() 函数向客户端发送原始的 HTTP 报头。

认识到一点很重要，即必须在任何实际的输出被发送之前调用 header() 函数（在 PHP 4 以及更高的版本中，您可以使用输出缓存来解决此问题）：

```
<html>
<?php
// 结果出错
// 在调用 header() 之前已存在输出
header('Location: http://www.example.com/');
?>
```

语法

```
header(string,replace,http_response_code)
```

参数	描述
string	必需。规定要发送的报头字符串。
replace	可选。指示该报头是否替换之前的报头，或添加第二个报头。默认是 true（替换）。false（允许相同类型的多个报头）。
http_response_code	可选。把 HTTP 响应代码强制为指定的值。（PHP 4 以及更高版本可用）

提示和注释

注释：从 PHP 4.4 之后，该函数防止一次发送多个报头。这是对头部注入攻击的保护措施。

例子

例子 1

```
<?php
// Date in the past
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
header("Cache-Control: no-cache");
header("Pragma: no-cache");
?>

<html>
<body>

...
...
```

注释：用户可能会设置一些选项来更改浏览器的默认缓存设置。通过发送上面的报头，您可以覆盖任何这些设置，强制浏览器不进行缓存！

例子 2

提示用户保存一个生成的 PDF 文件（Content-Disposition 报头用于提供一个推荐的文件名，并强制浏览器显示保存对话框）：

```
<?php
header("Content-type:application/pdf");

// 文件将被称为 downloaded.pdf
header("Content-Disposition:attachment;filename='downloaded.pdf'");

// PDF 源在 original.pdf 中
readfile("original.pdf");
?>

<html>
<body>

...
...
```

注释：微软 IE 5.5 存在一个阻止以上机制的 bug。通过升级为 Service Pack 2 或更高的版本，可以解决该 bug。

PHP headers_list() 函数

定义和用法

headers_list() 函数返回已发送的（或待发送的）响应头部的一个列表。

该函数返回包含报头的数组。

语法

```
headers_list()
```

提示和注释

提示：如需确定是否已发送报头，请使用 headers_sent() 函数。

例子

```
<?php
setcookie("TestCookie","SomeValue");
header("X-Sample-Test: foo");
header('Content-type: text/plain');
?>

<html>
<body>

<?php
// 发送哪些报头？
var_dump(headers_list());
?>

</body>
</html>
```

PHP headers_sent() 函数

定义和用法

headers_sent() 函数检查 HTTP 标头是否已被发送以及在哪里被发送。

如果报头已发送，则返回 true，否则返回 false。

语法

```
headers_sent(_file_,_line_)
```

参数	描述
file,line	可选。如果设置 *file* 和 *line* 参数，headers_sent() 会把输出开始的 PHP 源文件名和行号存入 file 和 line 变量中。

提示和注释

注释：一旦报头块已经发送，就不能使用 header() 函数 来发送其它的标头。使用此函数至少可以避免与 HTTP 标头有关的错误信息。

注释：可选的 *file* 和 *line* 参数是 PHP 4.3 中新加的。

例子

例子 1

```
<?php
// 如果报头未发送，则发送一个
if (!headers_sent())
  {
  header("Location: http://www.w3school.com.cn/");
  exit;
  }
?>

<html>
<body>

...
...
```

例子 2

使用可选的 file 和 line 参数：

```
<?php
// 传递 $file 和 $line，供日后使用
// 不要预先为它们赋值
if (!headers_sent($file, $line))
  {
  header("Location: http://www.w3school.com.cn/");
  exit;
  // Trigger an error here
  }
else
  {
  echo "Headers sent in $file on line $line";
  exit;
  }
?>

<html>
<body>

...
...
```

PHP setcookie() 函数

定义和用法

setcookie() 函数向客户端发送一个 HTTP cookie。

cookie 是由服务器发送到浏览器的变量。cookie 通常是服务器嵌入到用户计算机中的小文本文件。每当计算机通过浏览器请求一个页面，就会发送这个 cookie。

cookie 的名称指定为相同名称的变量。例如，如果被发送的 cookie 名为 "name"，会自动创建名为 $user 的变量，包含 cookie 的值。

必须在任何其他输出发送前对 cookie 进行赋值。

如果成功，则该函数返回 true，否则返回 false。

语法

```
setcookie(name,value,expire,path,domain,secure)
```

参数	描述
name	必需。规定 cookie 的名称。
value	必需。规定 cookie 的值。
expire	可选。规定 cookie 的有效期。
path	可选。规定 cookie 的服务器路径。
domain	可选。规定 cookie 的域名。
secure	可选。规定是否通过安全的 HTTPS 连接来传输 cookie。

提示和注释

注释：可以通过 $HTTP_COOKIE_VARS["user"] 或 $_COOKIE["user"] 来访问名为 "user" 的 cookie 的值。

注释：在发送 cookie 时，cookie 的值会自动进行 URL 编码。接收时会进行 URL 解码。如果你不需要这样，可以使用 setrawcookie() 代替。

例子

例子 1

设置并发送 cookie：

```
<?php
$value = "my cookie value";

// 发送一个简单的 cookie
setcookie("TestCookie",$value);
?>

<html>
<body>

...
...
```

```
<?php
$value = "my cookie value";

// 发送一个 24 小时候过期的 cookie
setcookie("TestCookie",$value, time()+3600*24);
?>

<html>
<body>

...
...
```

例子 2

检索 cookie 值的不同方法：

```
<html>
<body>

<?php

// 输出个别的 cookie
echo $_COOKIE["TestCookie"];
echo "<br />";
echo $HTTP_COOKIE_VARS["TestCookie"];
echo "<br />";

// 输出所有 cookie
print_r($_COOKIE);
?>

</body>
</html>
```

输出：

```
my cookie value
my cookie value
Array ([TestCookie] => my cookie value)
```

例子 3

通过把失效日期设置为过去的日期/时间，删除一个 cookie：

```
<?php
// 把失效日期设置为一小时前
setcookie ("TestCookie", "", time() - 3600);
?>

<html>
<body>

...
...
```

例子 4

创建一个数组 cookie：

```
<?php
setcookie("cookie[three]","cookiethree");
setcookie("cookie[two]","cookietwo");
setcookie("cookie[one]","cookieone");

// 输出 cookie （在重载页面后）
if (isset($_COOKIE["cookie"]))
  {
  foreach ($_COOKIE["cookie"] as $name => $value)
    {
    echo "$name : $value <br />";
    }
  }
?>

<html>
<body>

...
...
```

输出：

```
three : cookiethree
two : cookietwo
one : cookieone
```

PHP setrawcookie() 函数

定义和用法

setrawcookie() 函数不对 cookie 值进行 URL 编码，发送一个 HTTP cookie。

cookie 是由服务器发送到浏览器的变量。cookie 通常是服务器嵌入到用户计算机中的小文本文件。每当计算机通过浏览器请求一个页面，就会发送这个 cookie。

cookie 的名称指定为相同名称的变量。例如，如果被发送的 cookie 名为 "name"，会自动创建名为 $user 的变量，包含 cookie 的值。

必须在任何其他输出发送前对 cookie 进行赋值。

如果成功，则该函数返回 true，否则返回 false。

语法

```
setcookie(name,value,expire,path,domain,secure)
```

参数	描述
name	必需。规定 cookie 的名称。
value	必需。规定 cookie 的值。
expire	可选。规定 cookie 的有效期。
path	可选。规定 cookie 的服务器路径。
domain	可选。规定 cookie 的域名。
secure	可选。规定是否通过安全的 HTTPS 连接来传输 cookie。

提示和注释

注释：可以通过 $HTTP_COOKIE_VARS["user"] 或 $_COOKIE["user"] 来访问名为 "user" 的 cookie 的值。

注释：setrawcookie() 与 setcookie() 几乎完全相同，不同的是不会在发往客户机时，对 cookie 值进行自动 URL 编码。

例子

例子 1

设置并发送 cookie：

```
<?php
$value = "my cookie value";

// 发送一个简单的 cookie
setrawcookie("TestCookie",$value);
?>

<html>
<body>

...
...
```

```
<?php
$value = "my cookie value";

// 发送一个 24 小时候过期的 cookie
setrawcookie("TestCookie",$value, time()+3600*24);
?>

<html>
<body>

...
...
```

例子 2

检索 cookie 值的不同方法：

```
<html>
<body>

<?php

// 输出个别的 cookie
echo $_COOKIE["TestCookie"];
echo "<br />";
echo $HTTP_COOKIE_VARS["TestCookie"];
echo "<br />";

// 输出所有 cookie
print_r($_COOKIE);
?>

</body>
</html>
```

输出：

```
my cookie value
my cookie value
Array ([TestCookie] => my cookie value)
```

例子 3

通过把失效日期设置为过去的日期/时间，删除一个 cookie：

```
<?php
// 把失效日期设置为一小时前
setrawcookie ("TestCookie", "", time() - 3600);
?>

<html>
<body>

...
...
```

例子 4

创建一个数组 cookie：

```
<?php
setrawcookie("cookie[three]","cookiethree");
setrawcookie("cookie[two]","cookietwo");
setrawcookie("cookie[one]","cookieone");

// 输出 cookie （在重载页面后）
if (isset($_COOKIE["cookie"]))
  {
  foreach ($_COOKIE["cookie"] as $name => $value)
    {
    echo "$name : $value <br />";
    }
  }
?>

<html>
<body>

...
...
```

输出：

```
three : cookiethree
two : cookietwo
one : cookieone
```

PHP libxml 函数

PHP libxml 简介

libxml 函数和常量与 SimpleXML, XSLT 以及 DOM 一起使用。

安装

这些函数需要 libxml 程序包。在 xmlsoft.org 下载。

PHP libxml 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
libxml_clear_errors()	清空 libxml 错误缓冲。	5
libxml_get_errors()	检索错误数组。	5
libxml_get_last_error()	从 libxml 检索最后的错误。	5
libxml_set_streams_context()	为下一次 libxml 文档加载或写入设置流环境。	5
libxml_use_internal_errors()	禁用 libxml 错误，允许用户按需读取错误信息。	5

PHP libxml 常量

函数	描述	PHP
LIBXML_COMPACT	设置小型节点分配优化。会改善应用程序的性能。	5
LIBXML_DTDATTR	设置默认 DTD 属性。	5
LIBXML_DTDLOAD	加载外部子集。	5
LIBXML_DTDVALID	通过 DTD 进行验证。	5
LIBXML_NOBLANKS	删除空节点。	5
LIBXML_NOCDATA	把 CDATA 设置为文本节点。	5
LIBXML_NOEMPTYTAG	更改空标签（比如 改为 </br>）。仅在 DOMDocument->save() 和 DOMDocument->saveXML() 函数中可用。	5
LIBXML_NOENT	替代实体。	5
LIBXML_NOERROR	不显示错误报告。	5
LIBXML_NONET	在加载文档时停止网络访问。	5
LIBXML_NOWARNING	不显示警告报告。	5
LIBXML_NOXMLDECL	在保存文档时，撤销 XML 声明。	5
LIBXML_NSCLEAN	删除额外的命名空间声明。	5
LIBXML_XINCLUDE	使用 XInclude 置换。	5
LIBXML_ERR_ERROR	获得可恢复的错误。	5
LIBXML_ERR_FATAL	获得致命错误。	5
LIBXML_ERR_NONE	获得无错误。	5
LIBXML_ERR_WARNING	获得简单警告。	5
LIBXML_VERSION	获得 libxml 版本（例如：20605 或 20617）。	5
LIBXML_DOTTED_VERSION	获得有点号的 libxml 版本（例如：2.6.5 或 2.6.17）。	5

PHP libxml_clear_errors() 函数

定义和用法

libxml_clear_errors() 函数清空 libxml 错误缓冲。

语法

```
libxml_clear_errors()
```

例子

```
<?php
libxml_clear_errors()
?>
```

PHP libxml_get_errors() 函数

定义和用法

libxml_get_errors() 函数从 libxml 错误缓冲中获取错误。

该函数返回错误对象的一个数组，如果 libxml 错误缓冲中没有错误，则返回一个空数组。

语法

```
libxml_get_errors()
```

例子

```
<?php
libxml_get_errors()
?>
```

PHP libxml_get_last_error() 函数

定义和用法

libxml_get_last_error() 函数从 libxml 错误缓冲中获取最后一个错误。

若成功，则返回一个错误对象。若失败，或 libxml 错误缓冲中没有错误，则返回 false。

语法

```
libxml_get_last_error()
```

例子

```
<?php
libxml_get_last_error()
?>
```

PHP libxml_use_internal_errors() 函数

定义和用法

libxml_use_internal_errors() 函数禁用标准的 libxml 错误，并启用用户错误处理。

语法

```
libxml_use_internal_errors(user_errors)
```

参数	描述
user_errors	可选。规定是否应该启用用户错误处理。默认是 false。

说明

该函数返回 *user_errors* 参数之前的值。

例子

```
<?php
libxml_use_internal_errors()
?>
```

PHP Mail 函数

PHP Mail 简介

HTTP 函数允许您从脚本中直接发送电子邮件。

需求

要使邮件函数可用，PHP 需要已安装且正在运行的邮件系统。要使用的程序是由 php.ini 文件中的配置设置定义的。

安装

邮件函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

运行时配置

邮件函数的行为受 php.ini 的影响。

Mail 配置选项

名称	默认	描述	可更改
SMTP	"localhost"	Windows 专用：SMTP 服务器的 DNS 名称或 IP 地址。	PHP_INI_ALL
smtp_port	"25"	Windows 专用：SMTP 端口号。自 PHP 4.3 起可用。	PHP_INI_ALL
sendmail_from	NULL	Windows 专用：规定从 PHP 发送的邮件中使用的 "from" 地址。	PHP_INI_ALL
sendmail_path	NULL	Unix 系统专用：规定sendmail 程序的路径（通常 /usr/sbin/sendmail 或 /usr/lib/sendmail）	PHP_INI_SYSTEM

PHP Mail 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
ezmlm_hash()	计算 EZMLM 邮件列表系统所需的散列值。	3
mail()	允许您从脚本中直接发送电子邮件。	3

PHP mail() 函数

定义和用法

mail() 函数允许您从脚本中直接发送电子邮件。

如果邮件的投递被成功地接收，则返回 true，否则返回 false。

语法

```
mail(to,subject,message,headers,parameters)
```

参数	描述
to	必需。规定邮件的接收者。
subject	必需。规定邮件的主题。该参数不能包含任何换行字符。
message	必需。规定要发送的消息。
headers	必需。规定额外的报头，比如 From, Cc 以及 Bcc。
parameters	必需。规定 sendmail 程序的额外参数。

说明

在 *message* 参数规定的消息中，行之间必须以一个 LF（\n）分隔。每行不能超过 70 个字符。

（Windows 下）当 PHP 直接连接到 SMTP 服务器时，如果在一行开头发现一个句号，则会被删掉。要避免此问题，将单个句号替换成两个句号。

```
<?php
$text = str_replace("\n.", "\n..", $text);
?>
```

提示和注释

注释：您需要紧记，邮件投递被接受，并不意味着邮件到达了计划的目的地。

例子

例子 1

发送一封简单的邮件：

```
<?php

$txt = "First line of text\nSecond line of text";

// 如果一行大于 70 个字符，请使用 wordwrap()。
$txt = wordwrap($txt,70);

// 发送邮件
mail("somebody@example.com","My subject",$txt);
?>
```

例子 2

发送带有额外报头的 email：

```
<?php

$to = "somebody@example.com";
$subject = "My subject";
$txt = "Hello world!";
$headers = "From: webmaster@example.com" . "\r\n" .
"CC: somebodyelse@example.com";

mail($to,$subject,$txt,$headers);
?>
```

例子 3

发送一封 HTML email：

```
<?php

$to = "somebody@example.com, somebodyelse@example.com";
$subject = "HTML email";

$message = "
<html>
<head>
<title>HTML email</title>
</head>
<body>
<p>This email contains HTML Tags!</p>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
</tr>
<tr>
<td>John</td>
<td>Doe</td>
</tr>
</table>
</body>
</html>
";

// 当发送 HTML 电子邮件时，请始终设置 content-type
$headers = "MIME-Version: 1.0" . "\r\n";
$headers .= "Content-type:text/html;charset=iso-8859-1" . "\r\n";

// 更多报头
$headers .= 'From: <webmaster@example.com>' . "\r\n";
$headers .= 'Cc: myboss@example.com' . "\r\n";

mail($to,$subject,$message,$headers);
?>
```

PHP Math 函数

PHP Math 简介

数学 (Math) 函数能处理 integer 和 float 范围内的值。

安装

数学 (Math) 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP Math 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
abs()	绝对值。	3
acos()	反余弦。	3
acosh()	反双曲余弦。	4
asin()	反正弦。	3
asinh()	反双曲正弦。	4
atan()	反正切。	3
atan2()	两个参数的反正切。	3
atanh()	反双曲正切。	4
base_convert()	在任意进制之间转换数字。	3
bindec()	把二进制转换为十进制。	3
ceil()	向上舍入为最接近的整数。	3
cos()	余弦。	3
cosh()	双曲余弦。	4
decbin()	把十进制转换为二进制。	3
dechex()	把十进制转换为十六进制。	3
decoct()	把十进制转换为八进制。	3
deg2rad()	将角度转换为弧度。	3

exp()	返回 E^x 的值。	3
expm1()	返回 E^x - 1 的值。	4
floor()	向下舍入为最接近的整数。	3
fmod()	返回除法的浮点数余数。	4
getrandmax()	显示随机数最大的可能值。	3
hexdec()	把十六进制转换为十进制。	3
hypot()	计算直角三角形的斜边长度。	4
is_finite()	判断是否为有限值。	4
is_infinite()	判断是否为无限值。	4
is_nan()	判断是否为合法数值。	4
lcg_value()	返回范围为 (0, 1) 的一个伪随机数。	4
log()	自然对数。	3
log10()	以 10 为底的对数。	3
log1p()	返回 log(1 + number)。	4
max()	返回最大值。	3
min()	返回最小值。	3
mt_getrandmax()	显示随机数的最大可能值。	3
mt_rand()	使用 Mersenne Twister 算法返回随机整数。	3
mt_srand()	播种 Mersenne Twister 随机数生成器。	3
octdec()	把八进制转换为十进制。	3
pi()	返回圆周率的值。	3
pow()	返回 x 的 y 次方。	3
rad2deg()	把弧度数转换为角度数。	3
rand()	返回随机整数。	3
round()	对浮点数进行四舍五入。	3
sin()	正弦。	3
sinh()	双曲正弦。	4
sqrt()	平方根。	3
srand()	播下随机数发生器种子。	3
tan()	正切。	3
tanh()	双曲正切。	4

PHP Math 常量

常量名	常量名	常量值	PHP
M_E	e	2.718281828459045235	4
M_EULER	Euler 常量	0.57721566490153286061	5.2.0
M_LNPI	log_e(pi)	1.14472988584940017414	5.2.0
M_LN2	log_e 2	0.69314718055994530942	4
M_LN10	log_e 10	2.30258509299404568402	4
M_LOG2E	log_2 e	1.4426950408889634074	4
M_LOG10E	log_10 e	0.43429448190325182765	4
M_PI	Pi	3.14159265358979323846	3
M_PI_2	pi/2	1.57079632679489661923	4
M_PI_4	pi/4	0.78539816339744830962	4
M_1_PI	1/pi	0.31830988618379067154	4
M_2_PI	2/pi	0.63661977236758134308	4
M_SQRTPI	sqrt(pi)	1.77245385090551602729	5.2.0
M_2_SQRTPI	2/sqrt(pi)	1.12837916709551257390	4
M_SQRT1_2	1/sqrt(2)	0.70710678118654752440	4
M_SQRT2	sqrt(2)	1.41421356237309504880	4
M_SQRT3	sqrt(3)	1.73205080756887729352	5.2.0

PHP abs() 函数

定义和用法

abs() 函数返回一个数的绝对值。

语法

```
abs(x)
```

参数	描述
x	必需。一个数。

说明

返回参数 *x* 的绝对值。如果参数 *x* 是 float，则返回的类型也是 float，否则返回 integer（因为 float 通常比 integer 有更大的取值范围）。

例子

```
<?php
echo(abs(6.7));
echo(abs(-3));
echo(abs(3));
?>
```

输出：

```
6.7
3
3
```

PHP acos() 函数

定义和用法

acos() 函数返回一个数的反余弦。

语法

```
acos(x)
```

参数	描述
x	必需。一个数，范围在 -1 到 1 之间。

说明

返回 *x* 的反余弦值，单位是弧度。acos() 是 cos() 的反函数，它的意思是在 acos() 范围里的每个值都是 a==cos(acos(a)) 。

如果 *x* 的值在 -1 到 1 之外，则返回 -1.#IND。如果为 -1，则返回 PI 的值。

例子

在本例中，我们将计算不同值的反余弦：

```
<?php
echo(acos(0.64));
echo(acos(0));
echo(acos(-1));
echo(acos(1));
echo(acos(2));
?>
```

输出：

```
0.876298061168
1.57079632679
3.14159265359
0
-1.#IND
```

PHP acosh() 函数

定义和用法

acosh() 函数返回一个数的反双曲余弦。

语法

```
acosh(x)
```

参数	描述
x	必需。一个数。

说明

返回 *x* 的反双曲余弦值，即，其双曲余弦为 *x* 的那个值。

提示和注释

注释：本函数未在 Windows 平台下实现。

PHP asin() 函数

定义和用法

asin() 函数返回不同数值的反正弦，返回的结果是介于 -PI/2 与 PI/2 之间的弧度值。

语法

```
asin(x)
```

参数	描述
x	必需。一个数，范围在 -1 到 1 之间。

说明

返回 *x* 的反正弦值，单位是弧度。asin() 是 sin() 的反函数，它的意思是在 asin() 范围里的每个值都是 a==sin(asin(a))。

如果 *x* 的值在 -1 到 1 之外，则返回 -1.#IND。如果为 1，则返回 PI/2 的值。

例子

在本例中，我们将计算不同值的反正弦：

```
<?php
echo(asin(0.64));
echo(asin(0));
echo(asin(-1));
echo(asin(1));
echo(asin(2))
?>
```

输出：

```
0.694498265627
0
-1.57079632679
1.57079632679
-1.#IND
```

PHP asinh() 函数

定义和用法

asinh() 函数返回一个数的反双曲正弦。

语法

```
asinh(x)
```

参数	描述
x	必需。一个数。

说明

返回 *x* 的反双曲正弦值，即，其双曲正弦为 *x* 的那个值。

提示和注释

注释：本函数未在 Windows 平台下实现。

PHP atan() 和 atan2() 函数

定义和用法

atan() 函数返回一个数值的反正切，返回值介于 -PI/2 与 PI/2 之间。

atan2() 函数返回两个参数的反正切，返回值为弧度，其值在 -PI 和 PI 之间（包括 -PI 和 PI）。

语法

```
atan(x)

atan2(x,y)
```

参数	描述
x	必需。一个数。
y	必需。一个数。

说明

atan() 函数返回 *x* 的反正切值，单位是弧度。atan() 是 tan() 的反函数，它的意思是在 atan() 范围里的每个值都是 a==tan(atan(a))。

atan2() 函数计算两个变量 *x* 和 *y* 的反正切值。和计算 y / x 的反正切相似，不同的是两个参数的符号是用来确定结果的象限之外。

例子 1

本例计算不同值的反正切：

```
<?php
echo(atan(0.50));
echo(atan(-0.50));
echo(atan(5));
echo(atan(10));
echo(atan(-5));
echo(atan(-10))
?>
```

输出：

```
0.463647609001
-0.463647609001
1.37340076695
1.4711276743
-1.37340076695
-1.4711276743
```

例子 2

本例计算不同的变量 x 和 y 的反正切值：

```
<?php
echo(atan2(0.50,0.50));
echo(atan2(-0.50,-0.50));
echo(atan2(5,5));
echo(atan2(10,20));
echo(atan2(-5,-5));
echo(atan2(-10,10))
?>
```

输出：

```
0.785398163397
-2.35619449019
0.785398163397
0.463647609001
-2.35619449019
-0.785398163397
```

PHP atanh() 函数

定义和用法

atanh() 函数返回一个角度的反双曲正切。

语法

```
atanh(x)
```

参数	描述
x	必需。一个数。

说明

atanh() 函数返回 *x* 的反双曲正切值，即，其双曲正切为 *x* 的那个值。

提示和注释

注释：本函数未在 Windows 平台下实现。

PHP base_convert() 函数

定义和用法

base_convert() 函数在任意进制之间转换数字。

语法

```
base_convert(number,frombase,tobase)
```

参数	描述
number	必需。原始值。
frombase	必需。数字原来的进制。
tobase	必需。要转换的进制。

说明

返回一个字符串，包含 *number* 以 *tobase* 进制的表示。*number* 本身的进制由 *frombase* 指定。*frombase* 和 *tobase* 都只能在 2 和 36 之间（包括 2 和 36）。高于十进制的数字用字母 a-z 表示，例如 a 表示 10，b 表示 11 以及 z 表示 35。

例子 1

把八进制数转换为十进制数：

```
<?php
$oct = "0031";
$dec = base_convert($oct,8,10);
echo "八进制的 $oct 等于十进制的 $dec。";
?>
```

输出：

```
八进制的 0031 等于十进制的 25。
```

例子 2

把八进制数转换为十六进制数：

```
<?php
$oct = "364";
$hex = base_convert($oct,8,16);
echo "八进制的 $oct 等于十六进制的 $hex。";
?>
```

输出：

```
八进制的 364 等于十六进制的 f4。
```

PHP bindec() 函数

定义和用法

bindec() 函数把二进制转换为十进制。

语法

```
bindec(binary_string)
```

参数	描述
binary_string	必需。规定要转换的二进制数。

说明

返回 *binary_string* 参数所表示的二进制数的十进制等价值。

bindec() 函数将一个二进制数转换成 integer。可转换的最大的数为 31 位 1 或者说十进制的 2147483647。PHP 4.1.0 开始，该函数可以处理大数值，这种情况下，它会返回 float 类型。

例子

```
<?php
echo bindec("0011");
echo bindec("01");
echo bindec("11000110011");
echo bindec("111");
?>
```

输出：

```
3
1
1587
7
```

PHP ceil() 函数

定义和用法

ceil() 函数向上舍入为最接近的整数。

语法

```
ceil(x)
```

参数	描述
x	必需。一个数。

说明

返回不小于 *x* 的下一个整数，*x* 如果有小数部分则进一位。ceil() 返回的类型仍然是 float，因为 float 值的范围通常比 integer 要大。

例子

在本例中，我们将对不同的值应用 ceil() 函数：

```
<?php
echo(ceil(0.60);
echo(ceil(0.40);
echo(ceil(5);
echo(ceil(5.1);
echo(ceil(-5.1);
echo(ceil(-5.9));
?>
```

输出：

```
1
1
5
6
-5
-5
```

PHP cos() 函数

定义和用法

cos() 函数返回一个数的余弦。

语法

```
cos(x)
```

参数	描述
x	必需。一个数。

说明

cos() 返回参数 *x* 的余弦值。参数 *x* 的单位为弧度。

提示和注释

注释：cos() 返回的数值在 -1 和 1 之间。

例子

在本例中，我们将计算不同值的余弦：

```
<?php
echo(cos(3));
echo(cos(-3));
echo(cos(0));
echo(cos(M_PI));
echo(cos(2*M_PI));
?>
```

输出：

```
-0.9899924966004454
-0.9899924966004454
1
-1
1
```

PHP cosh() 函数

定义和用法

cosh() 函数返回一个数的双曲余弦。

语法

```
cosh(x)
```

参数	描述
x	必需。一个数。

说明

返回 *x* 的双曲余弦值，定义为 (exp(arg) + exp(-arg))/2。

例子

在本例中，我们将返回不同数的双曲余弦：

```
<?php
echo(cosh(3);
echo(cosh(-3);
echo(cosh(0);
echo(cosh(M_PI);
echo(cosh(2*M_PI));
?>
```

输出：

```
10.0676619958
10.0676619958
1
11.5919532755
267.746761484
```

PHP decbin() 函数

定义和用法

decbin() 函数把十进制转换为二进制。

语法

```
decbin(dec_number)
```

参数	描述
dec_number	必需。规定要转换的十进制数。

说明

返回一个字符串，包含有给定 *dec_number* 参数的二进制表示。所能转换的最大数值为十进制的 4294967295，其结果为 32 个 1 的字符串。

例子

```
<?php
echo decbin("3");
echo decbin("1");
echo decbin("1587");
echo decbin("7");
?>
```

输出：

```
11
1
11000110011
111
```

PHP dechex() 函数

定义和用法

dechex() 函数把十进制转换为十六进制。

语法

```
dechex(dec_number)
```

参数	描述
dec_number	必需。规定要转换的十进制数。

说明

返回一个字符串，包含有给定 *binary_string* 参数的十六进制表示。所能转换的最大数值为十进制的 4294967295，其结果为 "ffffffff"。

例子

```
<?php
echo dechex("30");
echo dechex("10");
echo dechex("1587");
echo dechex("70");
?>
```

输出：

```
1e
a
633
46
```

PHP decoct() 函数

定义和用法

decoct() 函数把十进制转换为八进制。

语法

```
decoct(dec_number)
```

参数	描述
dec_number	必需。规定要转换的十进制数。

说明

返回一个字符串，包含有给定 *dec_number* 参数的八进制表示。所能转换的最大数值为十进制的 4294967295，其结果为 "37777777777"。

例子

```
<?php
echo decoct("30");
echo decoct("10");
echo decoct("1587");
echo decoct("70");
?>
```

输出：

```
36
12
3063
106
```

PHP deg2rad() 函数

定义和用法

deg2rad() 函数将角度转换为弧度。

语法

```
deg2rad(degree_number)
```

参数	描述
degree_number	必需。规定要转换的角度。

说明

本函数把 *degree_number* 从角度转换成弧度。

例子 1

```
<?php
echo deg2rad("30");
echo deg2rad("10");
echo deg2rad("1587");
echo deg2rad("70");
?>
```

输出：

```
0.523598775598
0.174532925199
27.6983752292
1.2217304764
```

例子 2

```
<?php
$deg = 180;
$rad = deg2rad($deg);
echo "角度 $deg 等于弧度 $rad";
?>
```

输出：

```
角度 180 等于弧度 3.14159265359
```

PHP exp() 函数

定义和用法

exp() 函数计算 e 的指数。

语法

```
exp(x)
```

参数	描述
x	必需。一个数。

说明

返回 e 的 *x* 次方值。。

提示和注释

提示：用 'e' 作为自然对数的底 2.718282。

例子

在本例中，我们将对不同的数应用 exp() 函数：

```
<?php
echo(exp(1));
echo(exp(-1));
echo(exp(5));
echo(exp(10))
?>
```

输出：

```
2.718281828459045
0.36787944117144233
148.4131591025766
22026.465794806718
```

PHP expm1() 函数

定义和用法

expm1() 函数返回 exp(x) - 1，甚至当 number 的值接近零也能计算出准确结果。

语法

```
expm1(x)
```

参数	描述
x	必需。一个数。

说明

expm1() 返回 'exp(x) - 1'，甚至当 *x* 的值接近零也能计算出准确结果。但是当两个数值趋近于相等的时候， 'exp (x) - 1' 就会变得不太准确。

提示和注释

警告：本函数是实验性的。本函数的行为，包括函数名称以及其它任何关于本函数的文档可能会在没有通知的情况下随 PHP 以后的发布而改变。使用本函数风险自担。

提示：用 'e' 作为自然对数的底 2.718282。

注释：本函数未在 Windows 平台下实现。

例子

在本例中，我们将对不同的数应用 expm1() 函数：

```
<?php
echo(expm1(1));
echo(expm1(-1));
echo(expm1(5));
echo(expm1(10))
?>
```

PHP floor() 函数

定义和用法

floor() 函数向下舍入为最接近的整数。

语法

```
floor(x)
```

参数	描述
x	必需。一个数。

说明

返回不大于 *x* 的下一个整数，将 *x* 的小数部分舍去取整。floor() 返回的类型仍然是 float，因为 float 值的范围通常比 integer 要大。

例子

在本例中，我们将对不同的数应用 floor() 函数：

```
<?php
echo(floor(0.60));
echo(floor(0.40));
echo(floor(5));
echo(floor(5.1));
echo(floor(-5.1));
echo(floor(-5.9))
?>
```

输出：

```
0
0
5
5
-6
-6
```

PHP fmod() 函数

定义和用法

fmod() 函数返回除法的浮点数余数。

语法

```
fmod(x,y)
```

参数	描述
x	必需。一个数。
y	必需。一个数。

说明

返回被除数（*x*）除以除数（*y*）所得的浮点数余数。余数（r）的定义是：x = i * y + r，其中 *i* 是整数。如果 *y* 是非零值，则 *r* 和 *x* 的符号相同并且其数量值小于 *y* 。

例子

在本例中，我们将使用 fmod() 函数来返回 5/2 的余数：

```
<?php
$r = fmod(5,2);
echo $r
?>
```

输出：

```
1
```

PHP hexdec() 函数

定义和用法

hexdec() 函数把十六进制转换为十进制。

语法

```
hexdec(hex_string)
```

参数	描述
hex_string	必需。规定要转换的十六进制数。

说明

返回与 *hex_string* 参数所表示的十六进制数等值的的十进制数。hexdec() 将一个十六进制字符串转换为十进制数。所能转换的最大数值为 7fffffff，即十进制的 2147483647。PHP 4.1.0 开始，该函数可以处理大数字，这种情况下，它会返回 float 类型。

hexdec() 将遇到的所有非十六进制字符替换成 0。这样，所有左边的零都被忽略，但右边的零会计入值中。

例子

```
<?php
echo hexdec("1e");
echo hexdec("a");
echo hexdec("11ff");
echo hexdec("cceeff");
?>
```

输出：

```
30
10
4607
13430527
```

PHP hypot() 函数

定义和用法

hypot() 函数计算一直角三角形的斜边长度。

语法

```
hypot(x,y)
```

参数	描述
x	必需。边 x 的长度。
y	必需。边 y 的长度。

说明

hypot() 函数将会跟据直角三角形的两直解边长度 x 和 y 计算其斜边的长度。或者是从标点 (x , y) 到原点的距离。该函数的算法等同于 sqrt(x*x* + *y*y)。

例子

```
<?php
echo hypot(2,3);
echo hypot(3,6);
echo hypot(3,6);
echo hypot(1,3);
?>
```

输出：

```
3.60555127546
6.7082039325
6.7082039325
3.16227766017
```

PHP is_finite() 函数

定义和用法

is_finite() 函数判断是否为有限值。

语法

```
is_finite(x)
```

参数	描述
x	必需。规定要检查的值。

说明

如果 *x* 是本机平台上 PHP 浮点数所允许范围中的一个合法的有限值，则返回 true。

例子

```
<?php
echo is_finite(2);
echo is_finite(log(0));
echo is_finite(2000);
?>
```

输出：

```
1

1
```

PHP is_infinite() 函数

定义和用法

is_infinite() 判断是否为无限值。

语法

```
is_infinite(x)
```

参数	描述
x	必需。规定要检查的值。

说明

如果 *x* 为无穷大（正的或负的），例如 log(0) 的结果或者任何超出本平台的浮点数范围的值，则返回 true。

例子

```
<?php
echo is_infinite(2);
echo is_infinite(log(0));
echo is_infinite(2000);
?>
```

输出：

```
1
```

PHP is_nan() 函数

定义和用法

is_nan() 判断是否为合法数值。

语法

```
is_nan(x)
```

参数	描述
x	必需。规定要检查的值。

说明

如果 *x* 为“非数值”，例如 acos(1.01) 的结果，则返回 true。

例子

```
<?php
echo is_nan(200);
echo is_nan(acos(1.01));
?>
```

输出：

```
1
```

PHP lcg_value() 函数

定义和用法

lcg_value() 组合线性同余发生器。

语法

```
lcg_value()
```

说明

lcg_value() 返回范围为 (0, 1) 的一个伪随机数。本函数组合了周期为 2^31 - 85 和 2^31 - 249 的两个同余发生器。本函数的周期等于这两个素数的乘积。

例子

```
<?php
echo lcg_value();
?>
```

输出类似：

```
0.508212039328
```

PHP log() 函数

定义和用法

log() 返回自然对数。

语法

```
log(x,base)
```

参数	描述
x	必需。一个数。
base	可选。如果规定了该参数，则返回 log_{base}x。

说明

如果指定了可选的参数 *base*，log() 返回 log_{base}x ，否则 log() 返回参数 *x* 的自然对数。

注释：参数 base 自 PHP 4.3.0 开始可用。你可以计算任意以 b 为底 n 的对数，但其实使用的是数学等式：logb(n) = log(n)/log(b)，其中 log 是自然对数。

例子

```
<?php
echo lcg_value();
?>
```

输出类似：

```
0.508212039328
```

PHP log10() 函数

定义和用法

log10() 以 10 为底的对数。

语法

```
log10(x)
```

参数	描述
x	必需。一个数。

说明

返回参数 *x* 以 10 为底的对数。

提示和注释：

注释：如果参数 x 是负数，则返回 -1.#IND。

例子

在本例中，我们对不同的数应用 log10() 函数：

```
<?php
echo(log10(2.7183));
echo(log10(2));
echo(log10(1));
echo(log10(0));
echo(log10(-1));
?>
```

输出类似：

```
0.434297385125
0.301029995664
0
-1.#INF
-1.#IND
```

PHP log1p() 函数

定义和用法

log1p() 以返回 log(1 + x)，甚至当 x 的值接近零也能计算出准确结果。

语法

```
log1p(x)
```

参数	描述
x	必需。一个数。

说明

警告：本函数是实验性的。本函数的行为，包括函数名称以及其它任何关于本函数的文档可能会在没有通知的情况下随 PHP 以后的发布而改变。使用本函数风险自担。

PHP max() 函数

定义和用法

max() 返回最大值。

语法

```
max(x,y)
```

参数	描述
x	必需。一个数。
y	必需。一个数。

说明

max() 返回参数中数值最大的值。

如果仅有一个参数且为数组，max() 返回该数组中最大的值。如果第一个参数是整数、字符串或浮点数，则至少需要两个参数而 max() 会返回这些值中最大的一个。可以比较无限多个值。

提示和注释

注释：PHP 会将非数值的字符串当成 0，但如果这个正是最大的数值则仍然会返回一个字符串。如果多个参数都求值为 0 且是最大值，max() 会返回其中数值的 0，如果参数中没有数值的 0，则返回按字母表顺序最大的字符串。

例子

在本例中，我们将使用 max() 来返回两个指定的数中的最大值：

```
<?php
echo(max(5,7));
echo(max(-3,5));
echo(max(-3,-5));
echo(max(7.25,7.30));
?>
```

输出类似：

```
7
5
-3
7.3
```

PHP min() 函数

定义和用法

min() 返回最小值。

语法

```
min(x,y)
```

参数	描述
x	必需。一个数。
y	必需。一个数。

说明

min() 返回参数中数值最小的。

如果仅有一个参数且为数组，min() 返回该数组中最小的值。如果第一个参数是整数、字符串或浮点数，则至少需要两个参数而 min() 会返回这些值中最小的一个。可以比较无限多个值。

提示和注释

注释：PHP 会将非数值的 string 当成 0，但如果这个正是最小的数值则仍然会返回一个字符串。如果多个参数都求值为 0 且是最小值，min() 会返回按字母表顺序最小的字符串，如果其中没有字符串的话，则返回数值的 0。

例子

在本例中，我们将使用 min() 来返回两个指定的数中的最小值：

```
<?php
echo(min(5,7));
echo(min(-3,5));
echo(min(-3,-5));
echo(min(7.25,7.30));
?>
```

输出类似：

```
5
-3
-5
7.25
```

PHP mt_getrandmax() 函数

定义和用法

mt_getrandmax() 显示随机数的最大可能值。

语法

```
mt_getrandmax()
```

说明

返回调用 mt_rand() 所能返回的最大的随机数。

例子

```
<?php
echo mt_getrandmax();
?>
```

输出类似：

```
3147483649
```

PHP mt_rand() 函数

定义和用法

mt_rand() 使用 Mersenne Twister 算法返回随机整数。

语法

```
mt_rand(min,max)
```

说明

如果没有提供可选参数 *min* 和 *max*，mt_rand() 返回 0 到 RAND_MAX 之间的伪随机数。例如想要 5 到 15（包括 5 和 15）之间的随机数，用 mt_rand(5, 15)。

很多老的 libc 的随机数发生器具有一些不确定和未知的特性而且很慢。PHP 的 rand() 函数默认使用 libc 随机数发生器。mt_rand() 函数是非正式用来替换它的。该函数用了 Mersenne Twister 中已知的特性作为随机数发生器，它可以产生随机数值的平均速度比 libc 提供的 rand() 快四倍。

提示和注释

注释：自 PHP 4.2.0 起，不再需要用 srand() 或 mt_srand() 函数给随机数发生器播种，现在已自动完成。

注释：在 3.0.7 之前的版本中，max 的含义是 range 。要在这些版本中得到和上例相同 5 到 15 的随机数，简短的例子是 mt_rand (5, 11)。

例子

在本例中，我们会返回一些随机数：

```
<?php
echo(mt_rand());
echo(mt_rand());
echo(mt_rand(10,100));
?>
```

输出类似：

```
3150906288
513289678
35
```

PHP mt_srand() 函数

定义和用法

mt_srand() 播种 Mersenne Twister 随机数生成器。

语法

```
mt_srand(seed)
```

参数	描述
seed	必需。用 seed 来给随机数发生器播种。

说明

从 PHP 4.2.0 版开始，*seed* 参数变为可选项，当该项为空时，会被设为随时数。

提示和注释

注释：自 PHP 4.2.0 起，不再需要用 srand() 或 mt_srand() 函数给随机数发生器播种，现已自动完成。

例子

在本例中，我们将播种随机数生成器：

```
<?php
mt_srand(mktime());
echo(mt_rand());
?>
```

输出类似：

```
1132656473
```

PHP octdec() 函数

定义和用法

octdec() 函数把八进制转换为十进制。

语法

```
octdec(octal_string)
```

参数	描述
octal_string	必需。规定要转换的八进制数。

说明

返回 *octal_string* 参数所表示的八进制数的十进制等值。可转换的最大的数值为 17777777777 或十进制的 2147483647。从 PHP 4.1.0 开始，该函数可以处理大数字，这种情况下，它会返回 float 类型。

例子

```
<?php
echo octdec("36");
echo octdec("12");
echo octdec("3063");
echo octdec("106");
?>
```

输出类似：

```
30
10
1587
70
```

PHP pi() 函数

定义和用法

pi() 函数返回圆周率的值。

语法

```
pi()
```

说明

返回圆周率的近似值。返回值的 float 精度是由 php.ini 中的 precision 指令确定。默认值是 14。您也可以使用 M_PI 常量，该常量产生与 pi() 完全相同的结果。

例子

```
<?php
echo pi();
?>
```

输出类似：

```
3.14159265359
```

PHP pow() 函数

定义和用法

pow() 函数返回 x 的 y 次方。

语法

```
pow(x,y)
```

参数	描述
x	必需。一个数。
y	必需。一个数。

说明

返回 *x* 的 *y* 次方的幂。如果可能，本函数会返回 integer。

如果不能计算幂，将发出一条警告，pow() 将返回 false。PHP 4.2.0 版开始 pow() 不要产生任何的警告。

例子

```
<?php
echo pow(4,2);
echo pow(6,2);
echo pow(-6,2);
echo pow(-6,-2);
echo pow(-6,5.5);
?>
```

输出类似：

```
16
36
36
0.0277777777778
-1.#IND
```

PHP rad2deg() 函数

定义和用法

rad2deg() 函数把弧度数转换为角度数。

语法

```
rad2deg(radian_number)
```

参数	描述
radian_number	必需。规定要转换的弧度。

说明

本函数将 *radian_number* 从弧度转换为角度。

例子

```
<?php
$rad = M_PI;
$deg = rad2deg($rad);
echo "$rad radians is equal to $deg degrees";
?>
```

输出类似：

```
3.14159265359 radians is equal to 180 degrees
```

PHP rand() 函数

定义和用法

rand() 函数返回随机整数。

语法

```
rand(min,max)
```

参数	描述
min,max	可选。规定随机数产生的范围。

说明

如果没有提供可选参数 *min* 和 *max*，rand() 返回 0 到 RAND_MAX 之间的伪随机整数。例如，想要 5 到 15（包括 5 和 15）之间的随机数，用 rand(5, 15)。

提示和注释

注释：在某些平台下（例如 Windows）RAND_MAX 只有 32768。如果需要的范围大于 32768，那么指定 min 和 max 参数就可以生成大于 RAND_MAX 的数了，或者考虑用 mt_rand() 来替代它。

注释：自 PHP 4.2.0 起，不再需要用 srand() 或 mt_srand() 函数给随机数发生器播种，现在已自动完成。

注释：在 3.0.7 之前的版本中，max 的含义是 range 。要在这些版本中得到和上例相同 5 到 15 的随机数，简短的例子是 rand (5, 11)。

例子

本例会返回一些随机数：

```
<?php
echo(rand();
echo(rand();
echo(rand(10,100))
?>
```

输出：

```
17757
3794
97
```

PHP round() 函数

定义和用法

round() 函数对浮点数进行四舍五入。

语法

```
round(x,prec)
```

参数	描述
x	可选。规定要舍入的数字。
prec	可选。规定小数点后的位数。

说明

返回将 *x* 根据指定精度 *prec* （十进制小数点后数字的数目）进行四舍五入的结果。*prec* 也可以是负数或零（默认值）。

提示和注释

注释：PHP 默认不能正确处理类似 "12,300.2" 的字符串。

注释：*prec* 参数是在 PHP 4 中被引入的。。

例子

```
<?php
echo(round(0.60));
echo(round(0.50));
echo(round(0.49));
echo(round(-4.40));
echo(round(-4.60));
?>
```

输出：

```
1
1
0
-4
-5
```

PHP sin() 函数

定义和用法

sin() 函数返回一个数的正弦。

语法

```
sin(x)
```

参数	描述
x	必需。一个数。

说明

sin() 返回参数 *x* 的正弦值。参数 *x* 的单位为弧度。

提示和注释

注释：sin() 函数返回的数值介于 -1 和 1 之间。

例子

在本例中，我们将计算不同值的正弦：

```
<?php
echo(sin(3));
echo(sin(-3));
echo(sin(0));
echo(sin(M_PI));
echo(sin(M_PI_2))
?>
```

输出：

```
0.14112000806
-0.14112000806
0
1.22460635382E-016
1
```

PHP sinh() 函数

定义和用法

sinh() 函数返回一个数的双曲正弦。

语法

```
sinh(x)
```

参数	描述
x	必需。一个数字。

说明

返回 *x* 的双曲正弦值，定义为 (exp(arg) - exp(-arg))/2。

例子

在本例中，我们将返回不同数的双曲正弦：

```
<?php
echo(sinh(3));
echo(sinh(-3));
echo(sinh(0));
echo(sinh(M_PI));
echo(sinh(M_PI_2));
?>
```

输出：

```
10.0178749274
-10.0178749274
0
11.5487393573
2.30129890231
```

PHP sqrt() 函数

定义和用法

sqrt() 函数返回一个数的平方根。

语法

```
sqrt(x)
```

参数	描述
x	必需。一个数字。

说明

返回 *x* 的平方根。

提示和注释

注释：如果参数 *x* 是负数，则 sqrt() 函数返回 -1.#IND。

例子

在本例中，我们将返回不同数的平方根：

```
<?php
echo(sqrt(0));
echo(sqrt(1));
echo(sqrt(9));
echo(sqrt(0.64));
echo(sqrt(-9));
?>
```

输出：

```
0
1
3
0.8
-1.#IND
```

PHP srand() 函数

定义和用法

srand() 函数播下随机数发生器种子。

语法

```
srand(seed)
```

参数	描述
seed	可选。用 seed 播下随机数发生器种子。

说明

从 PHP 4.2.0 版开始，*seed* 参数变为可选项，当该项为空时，会被设为随时数。

提示和注释

注释：自 PHP 4.2.0 起，不再需要用 srand() 或 mt_srand() 函数给随机数发生器播种，现在已自动完成。

例子

在本例中，我们将播种随机数发生器：

```
<?php
srand(mktime());
echo(rand());
?>
```

输出：

```
23054
```

PHP tan() 函数

定义和用法

tan() 函数返回正切。

语法

```
tan(x)
```

参数	描述
x	必需。一个数。

说明

tan() 返回参数 *x* 的正切值。参数 *x* 的单位为弧度

例子

在本例中，我们将返回不同的数的正切：

```
<?php
echo(tan(M_PI_4));
echo(tan(0.50));
echo(tan(-0.50));
echo(tan(5));
echo(tan(10));
echo(tan(-5));
echo(tan(-10));
?>
```

输出：

```
1
0.546302489844
-0.546302489844
-3.38051500625
0.648360827459
3.38051500625
-0.648360827459
```

PHP tanh() 函数

定义和用法

tanh() 函数返回双曲正切。

语法

```
tanh(x)
```

参数	描述
x	必需。一个数。

说明

返回 *x* 的双曲正切值，定义为 sinh(arg)/cosh(arg)。

例子

在本例中，我们将返回不同的数的双曲正切：

```
<?php
echo(tanh(M_PI_4));
echo(tanh(0.50));
echo(tanh(-0.50));
echo(tanh(5));
echo(tanh(10));
echo(tanh(-5));
echo(tanh(-10))
?>
```

输出：

```
0.655794202633
0.46211715726
-0.46211715726
0.999909204263
0.999999995878
-0.999909204263
-0.999999995878
```

PHP 5 MySQLi 函数

PHP MySQLi 简介

PHP MySQLi = PHP MySQL Improved!

MySQLi 函数允许您访问 MySQL 数据库服务器。

注释：MySQLi 扩展被设计用于 MySQL 4.1.13 版本或更新的版本。

安装 / Runtime 配置

为了能够顺利使用 MySQLi 函数，您必须在编译 PHP 时添加对 MySQLi 扩展的支持。

MySQLi 扩展是在 PHP 5.0.0 版本中引进的。MySQL Native Driver 包含在 PHP 5.3.0 版本。

有关安装的详细信息，请访问： http://www.php.net/manual/en/mysqli.installation.php

有关运行配置的详细信息，请访问： http://www.php.net/manual/en/mysqli.configuration.php

PHP 5 MySQLi 函数

函数	描述
mysqli_affected_rows()	返回前一次 MySQL 操作所影响的记录行数。
mysqli_autocommit()	打开或关闭自动提交数据库修改。
mysqli_change_user()	更改指定数据库连接的用户。
mysqli_character_set_name()	返回数据库连接的默认字符集。
mysqli_close()	关闭先前打开的数据库连接。
mysqli_commit()	提交当前事务。
mysqli_connect_errno()	返回上一次连接错误的错误代码。
mysqli_connect_error()	返回上一次连接错误的错误描述。
mysqli_connect()	打开一个到 MySQL 服务器的新的连接。
mysqli_data_seek()	调整结果指针到结果集中的一个任意行。
mysqli_debug()	执行调试操作。
mysqli_dump_debug_info()	转储调试信息到日志中。
mysqli_errno()	返回最近调用函数的最后一个错误代码。

mysqli_error_list()	返回最近调用函数的错误列表。
mysqli_error()	返回最近调用函数的最后一个错误描述。
mysqli_fetch_all()	从结果集中取得所有行作为关联数组，或数字数组，或二者兼有。
mysqli_fetch_array()	从结果集中取得一行作为关联数组，或数字数组，或二者兼有。
mysqli_fetch_assoc()	从结果集中取得一行作为关联数组。
mysqli_fetch_field_direct()	从结果集中取得某个单一字段的 meta-data，并作为对象返回。
mysqli_fetch_field()	从结果集中取得下一字段，并作为对象返回。
mysqli_fetch_fields()	返回结果中代表字段的对象的数组。
mysqli_fetch_lengths()	返回结果集中当前行的每个列的长度。
mysqli_fetch_object()	从结果集中取得当前行，并作为对象返回。
mysqli_fetch_row()	从结果集中取得一行，并作为枚举数组返回。
mysqli_field_count()	返回最近查询的列数。
mysqli_field_seek()	把结果集中的指针设置为指定字段的偏移量。
mysqli_field_tell()	返回结果集中的指针的位置。
mysqli_free_result()	释放结果内存。
mysqli_get_charset()	返回字符集对象。
mysqli_get_client_info()	返回 MySQL 客户端库版本。
mysqli_get_client_stats()	返回有关客户端每个进程的统计。
mysqli_get_client_version()	将 MySQL 客户端库版本作为整数返回。
mysqli_get_connection_stats()	返回有关客户端连接的统计。
mysqli_get_host_info()	返回 MySQL 服务器主机名和连接类型。
mysqli_get_proto_info()	返回 MySQL 协议版本。
mysqli_get_server_info()	返回 MySQL 服务器版本。
mysqli_get_server_version()	将 MySQL 服务器版本作为整数返回。
mysqli_info()	返回有关最近执行查询的信息。
mysqli_init()	初始化 MySQLi 并返回 mysqli_real_connect() 使用的资源。
mysqli_insert_id()	返回最后一个查询中自动生成的 ID。
mysql_kill()	请求服务器杀死一个 MySQL 线程。
mysqli_more_results()	检查一个多查询是否有更多的结果。

mysqli_multi_query()	执行一个或多个针对数据库的查询。
mysqli_next_result()	为 mysqli_multi_query() 准备下一个结果集。
mysqli_num_fields()	返回结果集中字段的数量。
mysqli_num_rows()	返回结果集中行的数量。
mysqli_options()	设置额外的连接选项，用于影响连接行为。
mysqli_ping()	进行一个服务器连接，如果连接已断开则尝试重新连接。
mysqli_prepare()	准备执行一个 SQL 语句。
mysqli_query()	执行某个针对数据库的查询。
mysqli_real_connect()	打开一个到 MySQL 服务器的新的链接。
mysqli_real_escape_string()	转义在 SQL 语句中使用的字符串中的特殊字符。
mysqli_real_query()	执行 SQL 查询
mysqli_reap_async_query()	返回异步查询的结果。
mysqli_refresh()	刷新表或缓存，或者重置复制服务器信息。
mysqli_rollback()	回滚数据库中的当前事务。
mysqli_select_db()	更改连接的默认数据库。
mysqli_set_charset()	设置默认客户端字符集。
mysqli_set_local_infile_default()	撤销用于 load local infile 命令的用户自定义句柄。
mysqli_set_local_infile_handler()	设置用于 LOAD DATA LOCAL INFILE 命令的回滚函数。
mysqli_sqlstate()	返回最后一个 MySQL 操作的 SQLSTATE 错误代码。
mysqli_ssl_set()	用于创建 SSL 安全连接。
mysqli_stat()	返回当前系统状态。
mysqli_stmt_init()	初始化声明并返回 mysqli_stmt_prepare() 使用的对象。
mysqli_store_result()	传输最后一个查询的结果集。
mysqli_thread_id()	返回当前连接的线程 ID。
mysqli_thread_safe()	返回是否将客户端库编译成 thread-safe。
mysqli_use_result()	从上次使用 mysqli_real_query() 执行的查询中初始化结果集的检索。
mysqli_warning_count()	返回连接中的最后一个查询的警告数量。

PHP mysqli_affected_rows() 函数

实例

从不同的查询中输出所影响记录行数：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Perform queries and print out affected rows
mysqli_query($con,"SELECT * FROM Persons");
echo "Affected rows: " . mysqli_affected_rows($con);

mysqli_query($con,"DELETE FROM Persons WHERE Age>32");
echo "Affected rows: " . mysqli_affected_rows($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_affected_rows() 函数返回前一次 MySQL 操作（SELECT、INSERT、UPDATE、REPLACE、DELETE）所影响的记录行数。

语法

mysqli*affected_rows(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	一个 **> 0** 的整数表示所影响的记录行数。**0** 表示没有受影响的记录。**-1** 表示查询返回错误。
PHP 版本：	5+
:--	---

PHP mysqli_autocommit() 函数

实例

关闭自动提交，做一些查询，然后提交查询：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Set autocommit to off
mysqli_autocommit($con,FALSE);

// Insert some values
mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Peter','Griffin',35)");
mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Glenn','Quagmire',33)");

// Commit transaction
mysqli_commit($con);

// Close connection
mysqli_close($con);
?>
```

定义和用法

mysqli_autocommit() 函数开启或关闭自动提交数据库修改。

提示：请查看 mysqli_commit() 函数，用于提交指定数据库连接的当前事务。请查看 mysqli_rollback() 函数，用于回滚当前事务。

语法

mysqli*autocommit(_connection,mode);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。
mode	必需。如果设置为 FALSE，则表示关闭 auto-commit。如果设置为 TRUE，则表示开启 auto-commit（提交任何等待查询）。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_change_user() 函数

实例

改变指定数据库连接的用户：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");

// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Reset all and select a new database
mysqli_change_user($link, "my_user", "my_password", "my_test");

mysqli_close($con);
?>
```

定义和用法

mysqli_change_user() 函数改变指定数据库连接的用户，并设置当前数据库。

语法

mysqli*change_user(_connection,username,password,dbname);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。
username	必需。规定 MySQL 用户名。
password	必需。规定 MySQL 密码。
dbname	必需。规定要改变的新数据库。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_character_set_name() 函数

实例

返回数据库连接的默认字符集：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$charset=mysqli_character_set_name($con);
echo "Default character set is: " . $charset;

mysqli_close($con);
?>
```

定义和用法

mysqli_character_set_name() 函数返回数据库连接的默认字符集。

语法

mysqli*character_set_name(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	指定连接的默认字符集。
PHP 版本：	5+
:--	---

PHP mysqli_close() 函数

实例

关闭先前打开的数据库连接：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");

// ....some PHP code...

mysqli_close($con);
?>
```

定义和用法

mysqli_close() 函数关闭先前打开的数据库连接。

语法

mysqli*close(_connection*);

参数	描述
connection	必需。规定要关闭的 MySQL 连接。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_commit() 函数

实例

关闭自动提交，做一些查询，然后提交查询：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Set autocommit to off
mysqli_autocommit($con,FALSE);

// Insert some values
mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Peter','Griffin',35)");
mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Glenn','Quagmire',33)");

// Commit transaction
mysqli_commit($con);

// Close connection
mysqli_close($con);
?>
```

定义和用法

mysqli_commit() 函数提交指定数据库连接的当前事务。

提示：请查看 mysqli_autocommit() 函数，用于开启或关闭自动提交数据库修改。请查看 mysqli_rollback() 函数，用于回滚当前事务。

语法

mysqli*commit(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_connect_errno() 函数

实例

返回上一次连接错误的错误代码：

```
<?php
$con=mysqli_connect("localhost","wrong_user","my_password","my_db");
// Check connection
if (!$con)
{
die("Connection error: " . mysqli_connect_errno();
}
?>
```

定义和用法

mysqli_connect_errno() 函数返回上一次连接错误的错误代码。

语法

mysqli*connect_errno()*;_

技术细节

返回值：	返回错误代码值，如果没有错误发生则返回 **0**。
PHP 版本：	5+
:--	---

PHP mysqli_connect_error() 函数

实例

返回上一次连接错误的错误描述：

```
<?php
$con=mysqli_connect("localhost","wrong_user","my_password","my_db");
// Check connection
if (!$con)
{
die("Connection error: " . mysqli_connect_error();
}
?>
```

定义和用法

mysqli_connect_error() 函数返回上一次连接错误的错误描述。

语法

mysqli*connect_error()*;_

技术细节

返回值：	返回一个描述错误的字符串。如果没有错误发生则返回 **NULL**。
PHP 版本：	5+
:--	---

PHP mysqli_connect() 函数

实例

打开一个到 MySQL 服务器的新的连接：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");

// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}
?>
```

定义和用法

mysqli_connect() 函数打开一个到 MySQL 服务器的新的连接。

语法

mysqli*connect(_host,username,password,dbname,port,socket*);

参数	描述
host	可选。规定主机名或 IP 地址。
username	可选。规定 MySQL 用户名。
password	可选。规定 MySQL 密码。
dbname	可选。规定默认使用的数据库。
port	可选。规定尝试连接到 MySQL 服务器的端口号。
socket	可选。规定 socket 或要使用的已命名 pipe。

技术细节

返回值：	返回一个代表到 **MySQL** 服务器的连接的对象。
PHP 版本：	5+
:--	---

PHP mysqli_data_seek() 函数

实例

在结果集中寻找行号 15：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Seek to row number 15
mysqli_data_seek($result,14);

// Fetch row
$row=mysqli_fetch_row($result);

printf ("Lastname: %s Age: %sn", $row[0], $row[1]);

// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_data_seek() 函数调整结果指针到结果集中的一个任意行。

语法

mysqli*data_seek(_result,offset*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。
offset	必需。规定字段偏移。范围必须在 0 和 行总数 - 1 之间。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_debug() 函数

实例

在本地机上的 "/temp/client.trace" 中创建一个 trace 文件：

```
<?php
mysqli_debug("d:t:o,/temp/client.trace");
?>
```

定义和用法

mysqli_debug() 函数用于执行调试操作。

注释：为了使用该函数，您必须编译 MySQL 客户端库来支持调试。

语法

mysqli*debug(_message);*

参数	描述
message	必需。一个代表要执行的调试操作的字符串。

技术细节

返回值：	**TRUE**
PHP 版本：	5+
:--	---

PHP mysqli_dump_debug_info() 函数

实例

转储调试信息到日志中：

```
<?php
mysqli_dump_debug_info($con);
?>
```

定义和用法

mysqli_dump_debug_info() 函数转储调试信息到日志中。

语法

mysqli*dump_debug_info(_link);*

参数	描述
link	必需。一个由 mysqli_connect() 或 mysqli_init() 返回的连接标识符。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_errno() 函数

实例

返回最近调用函数的最后一个错误代码：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Perform a query, check for error
if (!mysqli_query($con,"INSERT INTO Persons (FirstName) VALUES ('Glenn')"))
{
echo("Errorcode: " . mysqli_errno($con));
}

mysqli_close($con);
?>
```

定义和用法

mysqli_errno() 函数返回最近调用函数的最后一个错误代码。

语法

mysqli*errno(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回错误代码值。如果没有错误发生则返回 **0**。
PHP 版本：	5+
:--	---

PHP mysqli_error_list() 函数

实例

返回最近调用函数的错误列表：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Perform a query, check for error
if (!mysqli_query($con,"INSERT INTO Persons (FirstName) VALUES ('Glenn')"))
{
print_r(mysqli_error_list($con));
}

mysqli_close($con);
?>
```

定义和用法

mysqli_error_list() 函数返回最近调用函数的错误列表。

语法

mysqli*error_list(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回错误列表。每个错误都是一个带有 **errno**（错误代码）、**error**（错误文本）和 **sqlstate** 的关联数组。
PHP 版本：	5.4+
:--	---

PHP mysqli_error() 函数

实例

返回最近调用函数的最后一个错误描述：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Perform a query, check for error
if (!mysqli_query($con,"INSERT INTO Persons (FirstName) VALUES ('Glenn')"))
{
echo("Error description: " . mysqli_error($con));
}

mysqli_close($con);
?>
```

定义和用法

mysqli_error() 函数返回最近调用函数的最后一个错误描述。

语法

mysqli*error(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个带有错误描述的字符串。如果没有错误发生则返回 ""。
PHP 版本：	5+
:--	---

PHP mysqli_fetch_all() 函数

实例

从结果集中取得所有行作为关联数组：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";
$result=mysqli_query($con,$sql);

// Fetch all
mysqli_fetch_all($result,MYSQLI_ASSOC);

// Free result set
mysqli_free_result($result);

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_all() 函数从结果集中取得所有行作为关联数组，或数字数组，或二者兼有。

注释：该函数只在带有 MySQL Native Driver 时可用。

语法

mysqli*fetch_all(_result,resulttype*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。
resulttype	可选。规定应该产生哪种类型的数组。可以是以下值中的一个： `MYSQLI_ASSOC` `MYSQLI_NUM` `MYSQLI_BOTH`

技术细节

返回值：	返回包含结果行的关联数组或数字数组。
PHP 版本：	5.3+
:--	---

PHP mysqli_fetch_array() 函数

实例

从结果集中取得一行作为数字数组或关联数组：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";
$result=mysqli_query($con,$sql)

// Numeric array
$row=mysqli_fetch_array($result,MYSQLI_NUM);
printf ("%s (%s)n",$row[0],$row[1]);

// Associative array
$row=mysqli_fetch_array($result,MYSQLI_ASSOC);
printf ("%s (%s)n",$row["Lastname"],$row["Age"]);

// Free result set
mysqli_free_result($result);

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_array() 函数从结果集中取得一行作为关联数组，或数字数组，或二者兼有。

注释：该函数返回的字段名是区分大小写的。

语法

mysqli*fetch_array(_result,resulttype*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。
resulttype	可选。规定应该产生哪种类型的数组。可以是以下值中的一个：`MYSQLI_ASSOC` `MYSQLI_NUM` `MYSQLI_BOTH`

技术细节

返回值：	返回与读取行匹配的字符串数组。如果结果集中没有更多的行则返回 **NULL**。
PHP 版本：	5+
:--	---

PHP mysqli_fetch_assoc() 函数

实例

从结果集中取得一行作为关联数组：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";
$result=mysqli_query($con,$sql)

// Associative array
$row=mysqli_fetch_assoc($result);
printf ("%s (%s)n",$row["Lastname"],$row["Age"]);

// Free result set
mysqli_free_result($result);

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_array() 函数从结果集中取得一行作为关联数组。

注释：该函数返回的字段名是区分大小写的。

语法

mysqli*fetch_assoc(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	返回代表读取行的关联数组。如果结果集中没有更多的行则返回 **NULL**。
PHP 版本：	5+
:--	---

PHP mysqli_fetch_field_direct() 函数

实例

返回结果集中某个单一字段（列）的 meta-data，并输出字段名称、表格和最大长度：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Get field information for "Age"
$fieldinfo=mysqli_fetch_field_direct($result,1);

printf("Name: %sn",$fieldinfo->name);
printf("Table: %sn",$fieldinfo->table);
printf("max. Len: %dn",$fieldinfo->max_length);

// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_field_direct() 函数从结果集中取得某个单一字段（列）的 meta-data，并作为对象返回。

语法

mysqli*fetch_field_direct(_result,fieldnr);*

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。
fieldnr	必需。规定字段号。必须介于 0 和 字段数-1 之间。

技术细节

返回值：	返回包含字段定义信息的对象。如果没有可用信息则返回 **FALSE**。该对象有下列属性： `name` - 列名 `orgname` - 原始的列名（如果指定了别名） `table` - 表名 `orgtable` - 原始的表名（如果指定了别名） `def` - 该字段的默认值 `max_length` - 字段的最大宽度 `length` - 在表定义中规定的字段宽度 `charsetnr` - 字段的字符集号 `flags` - 字段的位标志 `type` - 用于字段的数据类型 `decimals` - 整数字段，小数点后的位数
PHP 版本：	5+
:--	---

PHP mysqli_fetch_field() 函数

实例

返回结果集中下一字段（列），然后输出每个字段名称、表格和最大长度：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Get field information for all fields
while ($fieldinfo=mysqli_fetch_field($result))
{
printf("Name: %sn",$fieldinfo->name);
printf("Table: %sn",$fieldinfo->table);
printf("max. Len: %dn",$fieldinfo->max_length);
}
// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_field() 函数从结果集中取得下一字段（列），并作为对象返回。

语法

mysqli*fetch_field(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	返回包含字段定义信息的对象。如果没有可用信息则返回 **FALSE**。该对象有下列属性： `name` - 列名 `orgname` - 原始的列名（如果指定了别名） `table` - 表名 `orgtable` - 原始的表名（如果指定了别名） `def` - 保留作为默认值，当前总是为 **""** `db` - 数据库（在 **PHP 5.3.6** 中新增的） `catalog` - 目录名称，总是为 **"def"**（自 **PHP 5.3.6** 起） `max_length` - 字段的最大宽度 `length` - 在表定义中规定的字段宽度 `charsetnr` - 字段的字符集号 `flags` - 字段的位标志 `type` - 用于字段的数据类型 `decimals` - 整数字段，小数点后的位数
PHP 版本：	5+
:--	---

PHP mysqli_fetch_fields() 函数

实例

返回结果集中代表字段（列）的对象的数组，然后输出每个字段名称、表格和最大长度：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Get field information for all fields
$fieldinfo=mysqli_fetch_fields($result);

foreach ($fieldinfo as $val)
{
printf("Name: %sn",$val->name);
printf("Table: %sn",$val->table);
printf("max. Len: %dn",$val->max_length);
}
// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_fields() 函数返回结果集中代表字段（列）的对象的数组。

语法

mysqli*fetch_fields(_result);*

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	返回包含字段定义信息的对象。如果没有可用信息则返回 **FALSE**。该对象有下列属性： `name` - 列名 `orgname` - 原始的列名（如果指定了别名） `table` - 表名 `orgtable` - 原始的表名（如果指定了别名） `max_length` - 字段的最大宽度 `length` - 在表定义中规定的字段宽度 `charsetnr` - 字段的字符集号 `flags` - 字段的位标志 `type` - 用于字段的数据类型 `decimals` - 整数字段，小数点后的位数
PHP 版本：	5+
:--	---

PHP mysqli_fetch_lengths() 函数

实例

返回结果集中的字段长度：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT * FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
$row=mysqli_fetch_row($result);

// Display field lengths
foreach (mysqli_fetch_lengths($result) as $i=>$val)
{
printf("Field %2d has length: %2dn",$i+1,$val);
}

// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_lengths() 函数返回结果集中的字段长度。

语法

mysqli*fetch_lengths(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	如果整数表示每个字段（列）的长度则返回数组，如果发生错误则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_fetch_object() 函数

实例

返回结果集中的当前行，然后输出每个字段的值：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
while ($obj=mysqli_fetch_object($result))
{
printf("%s (%s)n",$obj->Lastname,$obj->Age);
}
// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_object() 函数从结果集中取得当前行，并作为对象返回。

注释：该函数返回的字段名是区分大小写的。

语法

mysqli*fetch_object(_result,classname,params);*

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。
classname	可选。规定要实例化的类名称，设置属性并返回。
params	可选。规定一个传给 *classname* 对象构造器的参数数组。

技术细节

返回值：	返回带有所取得行的字符串属性的对象。如果在结果集中没有更多的行则返回 **NULL**。
PHP 版本：	5+
:--	---
更新日志：	在 PHP 5.0.0 中新增了作为不同对象返回的功能。
:--	---

PHP mysqli_fetch_row() 函数

实例

从结果集中取得行：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Fetch one and one row
while ($row=mysqli_fetch_row($result))
{
printf ("%s (%s)n",$row[0],$row[1]);
}
// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_fetch_row() 函数从结果集中取得一行，并作为枚举数组返回。

语法

mysqli*fetch_row(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	返回一个与所取得行相对应的字符串数组。如果在结果集中没有更多的行则返回 **NULL**。
PHP 版本：	5+
:--	---

PHP mysqli_field_count() 函数

实例

假设我们有一个 "Friends" 表，其中有 3 个字段 20 行记录。返回最近查询的列数：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

mysqli_query($con,"SELECT * FROM Friends");
// Get number of columns - will always return 3
mysqli_field_count($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_field_count() 函数返回最近查询的列数。

语法

mysqli*field_count(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个表示结果集中列数的整数。
PHP 版本：	5+
:--	---

PHP mysqli_field_seek() 函数

实例

设置结果集中第一个字段（列）的字段指针，然后通过 mysqli_fetch_field() 获取字段信息并输出字段名称、表格和最大长度：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Get field info for 1st column ("Lastname")
mysqli_field_seek($result,0);
$fieldinfo=mysqli_fetch_field($result);

printf("Name: %sn",$fieldinfo->name);
printf("Table: %sn",$fieldinfo->table);
printf("max. Len: %dn",$fieldinfo->max_length);

// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_field_seek() 函数把字段指针设置为指定字段的偏移量。

语法

mysqli*field_seek(_result,fieldnr*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。
fieldnr	必需。规定字段号。必须介于 0 和 字段数-1 之间。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_field_tell() 函数

实例

取得所有字段的字段信息，然后通过 mysqli_field_tell() 取得当前字段并输出字段名称、表格和最大长度：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{

// Get field info for all fields
while ($fieldinfo=mysqli_fetch_field($result))
{
// Get field cursor position
$currentfield=mysqli_field_tell($result);

printf("Column %d:n", $currentfield);
printf("Name: %sn", $fieldinfo->name);
printf("Table: %sn", $fieldinfo->table);
}

// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_field_tell() 函数返回字段指针的位置。

语法

mysqli*field_tell(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	返回字段指针的当前偏移量。
PHP 版本：	5+
:--	---

PHP mysqli_free_result() 函数

实例

从结果集中取得行，然后释放结果内存：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Fetch one and one row
while ($row=mysqli_fetch_row($result))
{
printf ("%s (%s)n",$row[0],$row[1]);
}
// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_free_result() 函数释放结果内存。

语法

mysqli*free_result(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	没有返回值。
PHP 版本：	5+
:--	---

PHP mysqli_get_charset() 函数

实例

返回带有属性的字符集对象：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

var_dump(mysqli_get_charset($con));

mysqli_close($con);
?>
```

定义和用法

mysqli_get_charset() 函数返回字符集对象。

语法

mysqli*get_charset(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回带有下列属性的字符集对象： `charset` - 字符集名称 `collation` - 排序规则名称 `dir` - 被获取的目录字符集或者 "" `min_length` - 以字节计的最小字符长度 `max_length` - 以字节计的最大字符长度 `number` - 内部字符集数 `state` - 字符集状态
PHP 版本：	5.1+
:--	---

PHP mysqli_get_client_info() 函数

实例

返回 MySQL 客户端库版本：

```
<?php

echo mysqli_get_client_info();

?>
```

定义和用法

mysqli_get_client_info() 函数返回 MySQL 客户端库版本。

语法

mysqli*get_client_info(_connection*)*;*

参数	描述
connection	可选。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个表示 **MySQL** 客户端库版本的字符串。
PHP 版本：	5+
:--	---

PHP mysqli_get_client_stats() 函数

实例

返回有关客户端每个进程的统计：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

print_r(mysqli_get_client_stats());

mysqli_close($con);
?>
```

定义和用法

mysqli_get_client_stats() 函数返回有关客户端每个进程的统计。

语法

mysqli*get_client_stats()*;_

技术细节

返回值：	如果成功则返回一个带有客户端统计的数组，如果失败则返回 **FALSE**。
PHP 版本：	5.3+
:--	---

PHP mysqli_get_client_version() 函数

实例

将 MySQL 客户端库版本作为整数返回：

```
<?php
echo mysqli_get_client_version();
?>
```

定义和用法

mysqli_get_client_version() 函数将 MySQL 客户端库版本作为整数返回。

MySQL 客户端库版本将按照以下格式返回： 主要版本*10000* + 次要版本100 + 子版本。例如：5.1.0 将返回 50100。

语法

mysqli*get_client_version(_connection*);

参数	描述
connection	可选。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个表示 **MySQL** 客户端库版本的整数。
PHP 版本：	5+
:--	---

PHP mysqli_get_connection_stats() 函数

实例

返回有关客户端连接的统计：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

print_r(mysqli_get_connection_stats($con));

mysqli_close($con);
?>
```

定义和用法

mysqli_get_connection_stats() 函数返回有关客户端连接的统计。

语法

mysqli*get_connection_stats(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	如果成功则返回一个带有连接统计的数组，如果失败则返回 **FALSE**。
PHP 版本：	5.3+
:--	---

PHP mysqli_get_connection_stats() 函数

实例

返回有关客户端连接的统计：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

print_r(mysqli_get_connection_stats($con));

mysqli_close($con);
?>
```

定义和用法

mysqli_get_connection_stats() 函数返回有关客户端连接的统计。

语法

mysqli*get_connection_stats(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	如果成功则返回一个带有连接统计的数组，如果失败则返回 **FALSE**。
PHP 版本：	5.3+
:--	---

PHP mysqli_get_host_info() 函数

实例

返回 MySQL 服务器主机名和连接类型：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

echo mysqli_get_host_info($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_get_host_info() 函数返回 MySQL 服务器主机名和连接类型。

语法

mysqli*get_host_info(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个表示 **MySQL** 服务器主机名和连接类型的字符串。
PHP 版本：	5+
:--	---

PHP mysqli_get_proto_info() 函数

实例

返回 MySQL 协议版本：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

echo mysqli_get_proto_info($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_get_proto_info() 函数返回 MySQL 协议版本。

语法

mysqli*get_proto_info(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个表示 **MySQL** 协议版本的整数。
PHP 版本：	5+
:--	---

PHP mysqli_get_server_info() 函数

实例

返回 MySQL 服务器版本：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

echo mysqli_get_server_info($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_get_server_info() 函数返回 MySQL 服务器版本。

语法

mysqli*get_server_info(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个表示 **MySQL** 服务器版本的字符串。
PHP 版本：	5+
:--	---

PHP mysqli_get_server_version() 函数

实例

将 MySQL 服务器版本作为整数返回：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

echo mysqli_get_server_version($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_get_server_version() 函数将 MySQL 服务器版本作为整数返回。

MySQL 服务器版本将按照以下格式返回： 主要版本*10000* + 次要版本100 + 子版本。例如：5.1.0 将返回 50100。

语法

mysqli*get_server_version(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个表示 **MySQL** 服务器版本的整数。
PHP 版本：	5+
:--	---

PHP mysqli_info() 函数

实例

返回有关最近执行查询的信息：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Perform queries
$sql1="CREATE TABLE testPersons LIKE Persons"
mysqli_query($con,$sql1);
$sql2="INSERT INTO testPersons SELECT * FROM Persons ORDER BY LastName LIMIT 10"
mysqli_query($con,$sql2);

// Print info about most recently executed query
echo mysqli_info($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_info() 函数返回有关最近执行查询的信息。

该函数作用于下列查询类型：

- INSERT INTO...SELECT...
- INSERT INTO...VALUES (...),(...),(...)
- LOAD DATA INFILE ...
- ALTER TABLE ...
- UPDATE ...

语法

mysqli*info(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个字符串，包含有关最近执行查询的额外信息。
PHP 版本：	5+
:--	---

PHP mysqli_init() 函数

实例

mysqli_init() 函数的使用：

```
<?php
$con=mysqli_init();
if (!$con)
{
die("mysqli_init failed");
}

if (!mysqli_real_connect($con,"localhost","my_user","my_password","my_db"))
{
die("Connect Error: " . mysqli_connect_error());
}

mysqli_close($con);
?>
```

定义和用法

mysqli_init() 函数初始化 MySQLi 并返回 mysqli_real_connect() 使用的对象。

语法

mmysqli_init();

技术细节

返回值：	返回一个对象。
PHP 版本：	5+
:--	---

PHP mysqli_insert_id() 函数

实例

假设 Persons 表有一个自动生成的 ID 字段。返回最后一次查询中的 ID：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Glenn','Quagmire',33)");

// Print auto-generated id
echo "New record has id: " . mysqli_insert_id($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_insert_id() 函数返回最后一个查询中自动生成的 ID（通过 AUTO_INCREMENT 生成）。

语法

mysqli*insert_id(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个在最后一个查询中自动生成的带有 **AUTO_INCREMENT** 字段值的整数。如果数字 **>** 最大整数值，它将返回一个字符串。如果没有更新或没有 **AUTO_INCREMENT** 字段，将返回 **0**。
PHP 版本：	5+
:--	---

PHP mysqli_kill() 函数

实例

返回当前连接的线程 ID，然后杀死连接：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Get thread id
$t_id=mysqli_thread_id($con);

// Kill connection
mysqli_kill($con,$t_id);
?>
```

定义和用法

mysqli*kill() 函数请求服务器杀死一个由 _processid* 参数指定的 MySQL 线程。

语法

mysqli*kill(_connection,processid*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。
processid	必需。由 mysqli_thread_id() 返回的线程 ID。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_more_results() 函数

定义和用法

mysqli_more_results() 函数检查一个多查询是否有更多的结果。

语法

mysqli*more_results(_connection);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_multi_query() 函数

实例

执行多个针对数据库的查询：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql = "SELECT Lastname FROM Persons ORDER BY LastName;";
$sql .= "SELECT Country FROM Customers";

// Execute multi query
if (mysqli_multi_query($con,$sql))
{
do
{
// Store first result set
if ($result=mysqli_store_result($con))
{
while ($row=mysqli_fetch_row($result))
{
printf("%sn",$row[0]);
}
mysqli_free_result($con);
}
}
while (mysqli_next_result($con));
}

mysqli_close($con);
?>
```

定义和用法

mysqli_multi_query() 函数执行一个或多个针对数据库的查询。多个查询用分号进行分隔。

语法

mysqli*multi_query(_connection,query*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。
query	必需。规定一个或多个查询，用分号进行分隔。

技术细节

返回值：	如果第一个查询失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_next_result() 函数

实例

执行多个针对数据库的查询。请使用 mysqli_next_result() 函数来准备下一个结果集：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql = "SELECT Lastname FROM Persons ORDER BY LastName;";
$sql .= "SELECT Country FROM Customers";

// Execute multi query
if (mysqli_multi_query($con,$sql))
{
do
{
// Store first result set
if ($result=mysqli_store_result($con))
{
while ($row=mysqli_fetch_row($result))
{
printf("%sn",$row[0]);
}
mysqli_free_result($con);
}
}
while (mysqli_next_result($con));
}

mysqli_close($con);
?>
```

定义和用法

mysqli_next_result() 函数为 mysqli_multi_query() 准备下一个结果集。

语法

mysqli*next_result(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_num_fields() 函数

实例

返回结果集中字段（列）的数量：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Return the number of fields in result set
$fieldcount=mysqli_num_fields($result);
printf("Result set has %d fields.n",$fieldcount);
// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_num_fields() 函数返回结果集中字段（列）的数量。

语法

mysqli*num_fields(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	返回结果集中字段的数量。
PHP 版本：	5+
:--	---

PHP mysqli_num_rows() 函数

实例

返回结果集中行的数量：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$sql="SELECT Lastname,Age FROM Persons ORDER BY Lastname";

if ($result=mysqli_query($con,$sql))
{
// Return the number of rows in result set
$rowcount=mysqli_num_rows($result);
printf("Result set has %d rows.n",$rowcount);
// Free result set
mysqli_free_result($result);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_num_rows() 函数返回结果集中行的数量。

语法

mysqli*num_rows(_result*);

参数	描述
result	必需。规定由 mysqli_query()、mysqli_store_result() 或 mysqli_use_result() 返回的结果集标识符。

技术细节

返回值：	返回结果集中行的数量。
PHP 版本：	5+
:--	---

PHP mysqli_options() 函数

实例

打开一个到 MySQL 服务器的新连接：

```
<?php
$con=mysqli_init();
if (!$con)
{
die("mysqli_init failed");
}

mysqli_options($con,MYSQLI_READ_DEFAULT_FILE,"myfile.cnf");

if (!mysqli_real_connect($con,"localhost","my_user","my_password","my_db"))
{
die("Connect Error: " . mysqli_connect_error());
}

mysqli_close($con);
?>
```

定义和用法

mysqli_options() 函数设置额外的连接选项，用于影响连接行为。

mysqli_options() 函数可以被调用若干次来设置若干个选项。

注释：mysqli_options() 函数可以在 mysqli_init() 之后和 mysqli_real_connect() 之前被调用。

语法

mysqli*options(_connection,option,value);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。
option	必需。规定要设置的选项。可以是下列值中的一个： `MYSQLI_OPT_CONNECT_TIMEOUT` - 以秒为单位的连接超时时间 `MYSQLI_OPT_LOCAL_INFILE` - 启用/禁用 LOAD LOCAL INFILE `MYSQLI_INIT_COMMAND` - 在连接到 MySQL 服务器之后的执行命令 `MYSQLI_READ_DEFAULT_FILE` - 从已命名的文件而不是 my.cnf 中读取选项 `MYSQLI_READ_DEFAULT_GROUP` - 从 my.cnf 或者 MYSQLI_READ_DEFAULT_FILE 中指定的文件中的已命名组中读取选项 `MYSQLI_SERVER_PUBLIC_KEY` - 基于 SHA-256 认证的 RSA 公共密钥文件
value	必需。规定 *option* 的值。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---
更新日志：	在 PHP 5.5 中新增了 MYSQLI_SERVER_PUBLIC_KEY 选项。
:--	---

PHP mysqli_ping() 函数

实例

进行一个服务器连接：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");

// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Check if server is alive
if (mysqli_ping($con))
{
echo "Connection is ok!";
}
else
{
echo "Error: ". mysqli_error($con);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_ping() 函数进行一个服务器连接，如果连接已断开则尝试重新连接。

语法

mysqli*ping(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_query() 函数

实例

执行针对数据库的查询：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Perform queries
mysqli_query($con,"SELECT * FROM Persons");
mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Glenn','Quagmire',33)");

mysqli_close($con);
?>
```

定义和用法

mysqli_query() 函数执行某个针对数据库的查询。

语法

mysqli*query(_connection,query,resultmode*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。
query	必需，规定查询字符串。
resultmode	可选。一个常量。可以是下列值中的任意一个： `MYSQLI_USE_RESULT` （如果需要检索大量数据，请使用这个） `MYSQLI_STORE_RESULT` （默认）

技术细节

返回值：	针对成功的 **SELECT、SHOW、DESCRIBE 或 EXPLAIN** 查询，将返回一个 **mysqli_result** 对象。针对其他成功的查询，将返回 **TRUE**。如果失败，则返回 **FALSE**。
PHP 版本：	5+
:--	---
更新日志：	在 PHP 5.3.0 中新增了异步查询的功能。
:--	---

PHP mysqli_real_connect() 函数

实例

打开一个到 MySQL 服务器的新连接：

```
<?php
$con=mysqli_init();
if (!$con)
{
die("mysqli_init failed");
}

if (!mysqli_real_connect($con,"localhost","my_user","my_password","my_db"))
{
die("Connect Error: " . mysqli_connect_error());
}

mysqli_close($con);
?>
```

定义和用法

mysqli_real_connect() 函数打开一个到 MySQL 服务器的新连接。

mysqli_real_connect() 函数与 mysqli_connect() 函数在以下几个方面存在差异：

- mysqli_real_connect() 要求一个由 mysqli_init() 创建的有效的对象。
- mysqli_real_connect() 可以与 mysqli_options() 一同使用来设置连接的不同选项。
- mysqli_real_connect() 有一个 flag 参数。

语法

mysqli*real_connect(_connection,host,username,password,dbname,port,socket,flag*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。
host	可选。规定主机名或 IP 地址。
username	可选。规定 MySQL 用户名。
password	可选。规定 MySQL 密码。
dbname	可选。规定要使用的默认数据库。
port	可选。规定尝试连接到 MySQL 服务器的端口号。
socket	可选。规定 socket 或要使用的已命名 pipe。
flag	可选。规定不同的连接选项。可能的值： `MYSQLI_CLIENT_COMPRESS` - 使用压缩协议 `MYSQLI_CLIENT_FOUND_ROWS` - 返回匹配的行数（不是受影响的行数） `MYSQLI_CLIENT_IGNORE_SPACE` - 在函数名后允许空格，使函数名保留字 `MYSQLI_CLIENT_INTERACTIVE` - 在关闭连接之前允许不活动的 interactive_timeout 秒 `MYSQLI_CLIENT_SSL` - 使用 SSL 加密

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_real_escape_string() 函数

实例

转义字符串中的特殊字符：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");

// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

mysqli_query($con,"CREATE TABLE myPersons LIKE Persons");

$newpers="Da'Silva"

// This query will fail, cause we did not escape $newpers
mysqli_query($con,"INSERT into myPersons (Lastname) VALUES ('$newpers')");

$newpers=mysqli_real_escape_string($con,$newpers);

// This query will work, cause we escaped $newpers
mysqli_query($con,"INSERT into myPersons (Lastname) VALUES ('$newpers')");

mysqli_close($con);
?>
```

定义和用法

mysqli_real_escape_string() 函数转义在 SQL 语句中使用的字符串中的特殊字符。

语法

mysqli*real_escape_string(_connection,escapestring*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。
escapestring	必需。要转义的字符串。编码的字符是 NUL（ASCII 0）、\n、\r、\、'、" 和 Control-Z。

技术细节

返回值：	返回已转义的字符串。
PHP 版本：	5+
:--	---

PHP mysqli_refresh() 函数

定义和用法

mysqli_refresh() 函数刷新表或缓存，或者重置复制服务器信息。

语法

mysqli*refresh(_connection,options);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。
options	要刷新的选项。可以是下列中的一个或多个（用 OR 分隔）： `MYSQLI_REFRESH_GRANT` - 刷新授权表 `MYSQLI_REFRESH_LOG` - 刷新记录 `MYSQLI_REFRESH_TABLES` - 刷新表缓存 `MYSQLI_REFRESH_HOSTS` - 刷新主机缓存 `MYSQLI_REFRESH_STATUS` - 重置状态变量 `MYSQLI_REFRESH_THREADS` - 刷新线程缓存 `MYSQLI_REFRESH_SLAVE` - 重置主服务器信息，重启从服务器 `MYSQLI_REFRESH_MASTER` - 移除二进制日志索引中的二进制日志文件，并截断索引文件。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5.3+
:--	---

PHP mysqli_rollback() 函数

实例

关闭自动提交，做一些查询，提交查询，然后回滚当前事务：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Set autocommit to off
mysqli_autocommit($con,FALSE);

// Insert some values
mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Peter','Griffin',35)");
mysqli_query($con,"INSERT INTO Persons (FirstName,LastName,Age)
VALUES ('Glenn','Quagmire',33)");

// Commit transaction
mysqli_commit($con);

// Rollback transaction
mysqli_rollback($con);

// Close connection
mysqli_close($con);
?>
```

定义和用法

mysqli_rollback() 函数回滚指定数据库连接的当前事务。

提示：请查看 mysqli_commit() 函数，用于提交指定数据库连接的当前事务。请查看 mysqli_autocommit() 函数，用于开启或关闭自动提交数据库修改。

语法

mysqli*rollback(_connection);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_select_db() 函数

实例

更改连接的默认数据库：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// ...some PHP code for database "my_db"...

// Change database to "test"
mysqli_select_db($con,"test");

// ...some PHP code for database "test"...

mysqli_close($con);
?>
```

定义和用法

mysqli_select_db() 函数用于更改连接的默认数据库。

语法

mysqli*select_db(_connection,dbname);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。
dbname	必需，规定要使用的默认数据库。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_set_charset() 函数

实例

设置默认客户端字符集：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Change character set to utf8
mysqli_set_charset($con,"utf8");

mysqli_close($con);
?>
```

定义和用法

mysqli_set_charset() 函数规定当与数据库服务器进行数据传送时要使用的默认字符集。

注释：在 Windows 平台上使用该函数，您需要 MySQL 客户端库 4.1.11 或以上版本（MySQL 5.0 需要 5.0.6 或以上版本）。

语法

mysqli*set_charset(_connection,charset);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。
charset	必需。规定默认字符集。

技术细节

返回值：	如果成功则返回 **TRUE**，如果失败则返回 **FALSE**。
PHP 版本：	5.0.5+
:--	---

PHP mysqli_sqlstate() 函数

实例

返回最后一个 MySQL 操作的 SQLSTATE 错误代码：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Table Persons already exists, so we should get an error
$sql="CREATE TABLE Persons (Firstname VARCHAR(30),Lastname VARCHAR(30),Age INT)"
if (!mysqli_query($con,$sql))
{
echo "SQLSTATE error: ". mysqli_sqlstate($con);
}

// Close connection
mysqli_close($con);
?>
```

定义和用法

mysqli_sqlstate() 函数返回最后一个错误的 SQLSTATE 错误代码。

错误代码包含五个字符。"00000" 表明没有错误。值由 ANSI SQL 和 ODBC 指定。

语法

mysqli*sqlstate(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个包含最后一个错误的 **SQLSTATE** 错误代码的字符串。
PHP 版本：	5+
:--	---

PHP mysqli_ssl_set() 函数

实例

创建 SSL 连接：

```
<?php
$con=mysqli_init();
if (!$con)
{
die("mysqli_init failed");
}

mysqli_ssl_set($con,"key.pem","cert.pem","cacert.pem",NULL,NULL);

if (!mysqli_real_connect($con,"localhost","my_user","my_password","my_db"))
{
die("Connect Error: " . mysqli_connect_error());
}

// Some queries...

mysqli_close($con);
?>
```

定义和用法

mysqli_ssl_set() 函数用于创建 SSL 安全连接。然而，该函数只有在启用 OpenSSL 支持时才有效。

注释：该函数必须在 mysqli_real_connect() 之前调用。

注释：在 PHP 5.3.3 之前的版本，MySQL Native Driver 不支持 SSL。自 PHP 5.3+ 起，在微软 Windows 上默认启用 MySQL Native Driver。

语法

mysqli*ssl_set(_connection,key,cert,ca,capath,cipher);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。
key	必需。规定密钥文件的路径名。
cert	必需。规定认证文件的路径名。
ca	必需。规定认证授权文件的路径名。
capath	必需。规定包含 PEM 格式的可信 SSL CA 认证的目录的路径名。
cipher	必需。规定用于 SSL 加密的可用密码列表。

技术细节

返回值：	总是返回 **TRUE**。如果 **SSL** 安装不正确，当您尝试连接的时候，**mysqli_real_connect()** 将返回一个错误。
PHP 版本：	5+
:--	---

PHP mysqli_stat() 函数

实例

创建 SSL 连接：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

echo "System status: ". mysqli_stat($con);

mysqli_close($con);
?>
```

定义和用法

mysqli_stat() 函数返回当前系统状态。

语法

mysqli*stat(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个描述服务器状态的字符串。如果发生错误则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP mysqli_stmt_init() 函数

实例

初始化声明并返回 mysqli_stmt_prepare() 使用的对象：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$city="Sandnes";

// Create a prepared statement
$stmt=mysqli_stmt_init($con);

if (mysqli_stmt_prepare($stmt,"SELECT District FROM City WHERE Name=?"))
{

// Bind parameters
mysqli_stmt_bind_param($stmt,"s",$city);

// Execute query
mysqli_stmt_execute($stmt);

// Bind result variables
mysqli_stmt_bind_result($stmt,$district);

// Fetch value
mysqli_stmt_fetch($stmt);

printf("%s is in district %s",$city,$district);

// Close statement
mysqli_stmt_close($stmt);
}

mysqli_close($con);
?>
```

定义和用法

mysqli_stmt_init() 函数初始化声明并返回 mysqli_stmt_prepare() 使用的对象。

语法

mysqli*stmt_init(_connection);*

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回一个对象。
PHP 版本：	5+
:--	---

PHP mysqli_thread_id() 函数

实例

返回当前连接的线程 ID，然后杀死连接：

```
<?php
$con=mysqli_connect("localhost","my_user","my_password","my_db");
// Check connection
if (mysqli_connect_errno($con))
{
echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

// Get thread id
$t_id=mysqli_thread_id($con);

// Kill connection
mysqli_kill($con,$t_id);
?>
```

定义和用法

mysqli_thread_id() 函数返回当前连接的线程 ID，然后使用 mysqli_kill() 函数杀死该连接。

注释：如果连接被损坏且重新连接，线程 ID 将会改变。因此，仅当您需要的时候才获取线程 ID。

语法

mysqli*thread_id(_connection*);

参数	描述
connection	必需。规定要使用的 MySQL 连接。

技术细节

返回值：	返回当前连接的线程 **ID**。
PHP 版本：	5+
:--	---

PHP mysqli_thread_safe() 函数

定义和用法

mysqli_thread_safe() 函数返回是否将客户端库编译成 thread-safe。

语法

mysqli*thread_safe()*;_

技术细节

返回值：	如果客户端库是 **thread-safe** 则返回 **TRUE**，否则返回 **FALSE**。
PHP 版本：	5+
:--	---

PHP PDO

PHP 数据对象 （PDO） 扩展为PHP访问数据库定义了一个轻量级的一致接口。

PDO 提供了一个数据访问抽象层，这意味着，不管使用哪种数据库，都可以用相同的函数（方法）来查询和获取数据。

PDO随PHP5.1发行，在PHP5.0的PECL扩展中也可以使用，无法运行于之前的PHP版本。

PDO 安装

你可以通过 PHP 的 phpinfo() 函数来查看是否安装了PDO扩展。

在 Unix 系统上安装 PDO

在Unix上或Linux上你需要添加以下扩展：

```
extension=pdo.so
```

Windows 用户

PDO 和所有主要的驱动作为共享扩展随 PHP 一起发布，要激活它们只需简单地编辑 php.ini 文件，并添加以下扩展：

```
extension=php_pdo.dll
```

除此之外还有以下对应的各种数据库扩展：

```
;extension=php_pdo_firebird.dll
;extension=php_pdo_informix.dll
;extension=php_pdo_mssql.dll
;extension=php_pdo_mysql.dll
;extension=php_pdo_oci.dll
;extension=php_pdo_oci8.dll
;extension=php_pdo_odbc.dll
;extension=php_pdo_pgsql.dll
;extension=php_pdo_sqlite.dll
```

在设定好这些配置后，我们需要重启PHP 或 Web服务器。

接下来我们们来看下具体的实例，以下为使用PDO连接MySql数据库的实例：

```
<?php
$dbms='mysql';     //数据库类型
$host='localhost'; //数据库主机名
$dbName='test';    //使用的数据库
$user='root';      //数据库连接用户名
$pass='';          //对应的密码
$dsn="$dbms:host=$host;dbname=$dbName";

try {
    $dbh = new PDO($dsn, $user, $pass); //初始化一个PDO对象
    echo "连接成功<br/>";
    /*你还可以进行一次搜索操作
    foreach ($dbh->query('SELECT * from FOO') as $row) {
        print_r($row); //你可以用 echo($GLOBAL); 来看到这些值
    }
    */
    $dbh = null;
} catch (PDOException $e) {
    die ("Error!: " . $e->getMessage() . "<br/>");
}
//默认这个不是长连接，如果需要数据库长连接，需要最后加一个参数：array(PDO::ATTR_PERSISTENT => true)
$db = new PDO($dsn, $user, $pass, array(PDO::ATTR_PERSISTENT => true));

?>
```

很简单吧，接下来就让我们来具体看下PHP PDO具体说明：

- 预定义常量
- PDO 类：
  - PDO::beginTransaction — 启动一个事务
  - PDO::commit — 提交一个事务
  - PDO::__construct — 创建一个表示数据库连接的 PDO 实例
  - PDO::errorCode — 获取跟数据库句柄上一次操作相关的 SQLSTATE
  - PDO::errorInfo — 返回最后一次操作数据库的错误信息
  - PDO::exec — 执行一条 SQL 语句，并返回受影响的行数
  - PDO::getAttribute — 取回一个数据库连接的属性
  - PDO::getAvailableDrivers — 返回一个可用驱动的数组
  - PDO::inTransaction — 检查是否在一个事务内
  - PDO::lastInsertId — 返回最后插入行的ID或序列值
  - PDO::prepare — 备要执行的SQL语句并返回一个 PDOStatement 对象
  - PDO::query — 执行 SQL 语句，返回PDOStatement对象,可以理解为结果集
  - PDO::quote — 为SQL语句中的字符串添加引号。
  - PDO::rollBack — 回滚一个事务
  - PDO::setAttribute — 设置属性
- PDOStatement 类：
  - PDOStatement::bindColumn — 绑定一列到一个 PHP 变量
  - PDOStatement::bindParam — 绑定一个参数到指定的变量名
  - PDOStatement::bindValue — 把一个值绑定到一个参数

- PDOStatement::closeCursor — 关闭游标，使语句能再次被执行。
- PDOStatement::columnCount — 返回结果集中的列数
- PDOStatement::debugDumpParams — 打印一条 SQL 预处理命令
- PDOStatement::errorCode — 获取跟上一次语句句柄操作相关的 SQLSTATE
- PDOStatement::errorInfo — 获取跟上一次语句句柄操作相关的扩展错误信息
- PDOStatement::execute — 执行一条预处理语句
- PDOStatement::fetch — 从结果集中获取下一行
- PDOStatement::fetchAll — 返回一个包含结果集中所有行的数组
- PDOStatement::fetchColumn — 从结果集中的下一行返回单独的一列。
- PDOStatement::fetchObject — 获取下一行并作为一个对象返回。
- PDOStatement::getAttribute — 检索一个语句属性
- PDOStatement::getColumnMeta — 返回结果集中一列的元数据
- PDOStatement::nextRowset — 在一个多行集语句句柄中推进到下一个行集
- PDOStatement::rowCount — 返回受上一个 SQL 语句影响的行数
- PDOStatement::setAttribute — 设置一个语句属性
- PDOStatement::setFetchMode — 为语句设置默认的获取模式。

PHP PDO预定义常量

以下常量由本扩展模块定义，因此只有在本扩展的模块被编译到PHP中，或者在运行时被动态加载后才有效。

注意： PDO使用类常量自PHP 5.1。以前的版本使用的全局常量形式PDO_PARAM_BOOL中。

PDO::PARAM_BOOL (integer)	
PDO::PARAM_NULL (integer)	表示 SQL 中的 NULL 数据类型。
PDO::PARAM_INT (integer)	表示 SQL 中的整型。
PDO::PARAM_STR (integer)	表示 SQL 中的 CHAR、VARCHAR 或其他字符
PDO::PARAM_LOB (integer)	表示 SQL 中大对象数据类型。
PDO::PARAM_STMT (integer)	表示一个记录集类型。当前尚未被任何驱动支
PDO::PARAM_INPUT_OUTPUT (integer)	指定参数为一个存储过程的 INOUT 参数。必须
PDO::FETCH_LAZY (integer)	指定获取方式，将结果集中的每一行作为一个 名。在 PDOStatement::fetchAll() 中无效。
PDO::FETCH_ASSOC (integer)	指定获取方式，将对应结果集中的每一行作为 PDO::FETCH_ASSOC每个列名只返回一个值
PDO::FETCH_NAMED (integer)	指定获取方式，将对应结果集中的每一行作为 PDO::FETCH_ASSOC每个列名 返回一个包含
PDO::FETCH_NUM (integer)	指定获取方式，将对应结果集中的每一行作为
PDO::FETCH_BOTH (integer)	指定获取方式，将对应结果集中的每一行作为
PDO::FETCH_OBJ (integer)	指定获取方式，将结果集中的每一行作为一个
PDO::FETCH_BOUND (integer)	指定获取方式，返回 TRUE 且将结果集中的列 PHP 变量。
PDO::FETCH_COLUMN (integer)	指定获取方式，从结果集中的下一行返回所需
PDO::FETCH_CLASS (integer)	指定获取方式，返回一个所请求类的新实例， 方法
PDO::FETCH_INTO (integer)	指定获取方式，更新一个请求类的现有实例，
PDO::FETCH_FUNC (integer)	允许在运行中完全用自定义的方式处理数据。
PDO::FETCH_GROUP (integer)	根据值分组返回。通常和 PDO::FETCH_COL
PDO::FETCH_UNIQUE (integer)	只取唯一值。
PDO::FETCH_KEY_PAIR (integer)	获取一个有两列的结果集到一个数组，其中第
PDO::FETCH_CLASSTYPE (integer)	根据第一列的值确定类名。

PDO::FETCH_SERIALIZE (integer)	类似 PDO::FETCH_INTO，但是以一个序列化 构造函数从不会被调用。
PDO::FETCH_PROPS_LATE (integer)	设置属性前调用构造函数。自 PHP 5.2.0 起可
PDO::ATTR_AUTOCOMMIT (integer)	如果此值为 FALSE，PDO 将试图禁用自动提
PDO::ATTR_PREFETCH (integer)	设置预取大小来为你的应用平衡速度和内存使 时也会占用更多的内存。
PDO::ATTR_TIMEOUT (integer)	设置连接数据库的超时秒数。
PDO::ATTR_ERRMODE (integer)	关于此属性的更多信息请参见 错误及错误处理
PDO::ATTR_SERVER_VERSION (integer)	此为只读属性；返回 PDO 所连接的数据库服
PDO::ATTR_CLIENT_VERSION (integer)	此为只读属性；返回 PDO 驱动所用客户端库
PDO::ATTR_SERVER_INFO (integer)	此为只读属性。返回一些关于 PDO 所连接的
PDO::ATTR_CONNECTION_STATUS (integer)	
PDO::ATTR_CASE (integer)	用类似 PDO::CASE_* 的常量强制列名为指定
PDO::ATTR_CURSOR_NAME (integer)	获取或设置使用游标的名称。当使用可滚动游
PDO::ATTR_CURSOR (integer)	选择游标类型。 PDO 当前支持 PDO::CURSO 确实需要一个可滚动游标。
PDO::ATTR_DRIVER_NAME (string)	返回驱动名称。使用 PDO::ATTR_DRIVER_N 子： `if ($db->getAttribute(PDO::ATTR_DRIV`
PDO::ATTR_ORACLE_NULLS (integer)	在获取数据时将空字符串转换成 SQL 中的 NU
PDO::ATTR_PERSISTENT (integer)	请求一个持久连接，而非创建一个新连接。关
PDO::ATTR_STATEMENT_CLASS (integer)	
PDO::ATTR_FETCH_CATALOG_NAMES (integer)	将包含的目录名添加到结果集中的每个列名前 支持此属性。
PDO::ATTR_FETCH_TABLE_NAMES (integer)	将包含的表名添加到结果集中的每个列名前面 此属性。
PDO::ATTR_STRINGIFY_FETCHES (integer)	
PDO::ATTR_MAX_COLUMN_LEN (integer)	
PDO::ATTR_DEFAULT_FETCH_MODE (integer)	自 PHP 5.2.0 起可用。
PDO::ATTR_EMULATE_PREPARES (integer)	自 PHP 5.1.3 起可用。
PDO::ERRMODE_SILENT (integer)	如果发生错误，则不显示错误或异常。希望开

PDO::ERRMODE_WARNING (integer)	如果发生错误，则显示一个 PHP E_WARNIN
PDO::ERRMODE_EXCEPTION (integer)	如果发生错误，则抛出一个 PDOException 异
PDO::CASE_NATURAL (integer)	保留数据库驱动返回的列名。
PDO::CASE_LOWER (integer)	强制列名小写。
PDO::CASE_UPPER (integer)	强制列名大写。
PDO::NULL_NATURAL (integer)	
PDO::NULL_EMPTY_STRING (integer)	
PDO::NULL_TO_STRING (integer)	
PDO::FETCH_ORI_NEXT (integer)	在结果集中获取下一行。仅对可滚动游标有效
PDO::FETCH_ORI_PRIOR (integer)	在结果集中获取上一行。仅对可滚动游标有效
PDO::FETCH_ORI_FIRST (integer)	在结果集中获取第一行。仅对可滚动游标有效
PDO::FETCH_ORI_LAST (integer)	在结果集中获取最后一行。仅对可滚动游标有
PDO::FETCH_ORI_ABS (integer)	根据行号从结果集中获取需要的行。仅对可滚
PDO::FETCH_ORI_REL (integer)	根据当前游标位置的相对位置从结果集中获取
PDO::CURSOR_FWDONLY (integer)	创建一个只进游标的 PDOStatement 对象。此
PDO::CURSOR_SCROLL (integer)	创建一个可滚动游标的 PDOStatement 对象。
PDO::ERR_NONE (string)	对应 SQLSTATE '00000'，表示 SQL 语句没有 否有错误发生时，此常量非常方便。在检查上
PDO::PARAM_EVT_ALLOC (integer)	分配事件
PDO::PARAM_EVT_FREE (integer)	解除分配事件
PDO::PARAM_EVT_EXEC_PRE (integer)	执行一条预处理语句之前触发事件。
PDO::PARAM_EVT_EXEC_POST (integer)	执行一条预处理语句之后触发事件。
PDO::PARAM_EVT_FETCH_PRE (integer)	从一个结果集中取出一条结果之前触发事件。
PDO::PARAM_EVT_FETCH_POST (integer)	从一个结果集中取出一条结果之后触发事件。
PDO::PARAM_EVT_NORMALIZE (integer)	在绑定参数注册允许驱动程序正常化变量名时

PHP PDO连接

连接是通过创建 PDO 基类的实例而建立的。不管使用哪种驱动程序，都是用 PDO 类名。

连接到 MySQL

```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
?>
```

注意：如果有任何连接错误，将抛出一个 PDOException 异常对象。

处理连接错误

```
<?php
try {
    $dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
    foreach($dbh->query('SELECT * from FOO') as $row) {
        print_r($row);
    }
    $dbh = null;
} catch (PDOException $e) {
    print "Error!: " . $e->getMessage() . "<br/>";
    die();
}
?>
```

连接数据成功后，返回一个 PDO 类的实例给脚本，此连接在 PDO 对象的生存周期中保持活动。

要想关闭连接，需要销毁对象以确保所有剩余到它的引用都被删除，可以赋一个 NULL 值给对象变量。

如果不这么做，PHP 在脚本结束时会自动关闭连接。

关闭一个连接:

```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
// 在此使用连接

// 现在运行完成，在此关闭连接
$dbh = null;
?>
```

很多 web 应用程序通过使用到数据库服务的持久连接获得好处。

持久连接在脚本结束后不会被关闭，且被缓存，当另一个使用相同凭证的脚本连接请求时被重用。

持久连接缓存可以避免每次脚本需要与数据库回话时建立一个新连接的开销，从而让 web 应用程序更快。

持久化连接

```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass, array(
    PDO::ATTR_PERSISTENT => true
));
?>
```

注意：如果想使用持久连接，必须在传递给 PDO 构造函数的驱动选项数组中设置 PDO::ATTR_PERSISTENT 。如果是在对象初始化之后用 PDO::setAttribute() 设置此属性，则驱动程序将不会使用持久连接。

PHP PDO 事务与自动提交

现在通过 PDO 连接上了，在开始进行查询前，必须先理解 PDO 是如何管理事务的。

事务支持四大特性（ACID）：

- 原子性（Atomicity）
- 一致性（Consistency）
- 隔离性（Isolation）
- 持久性（Durability）

通俗地讲，在一个事务中执行的任何操作，即使是分阶段执行的，也能保证安全地应用于数据库，并在提交时不会受到来自其他连接的干扰。

事务操作也可以根据请求自动撤销（假设还没有提交），这使得在脚本中处理错误更加容易。

事务通常是通过把一批更改"积蓄"起来然后使之同时生效而实现的；这样做的好处是可以大大地提供这些更改的效率。

换句话说，事务可以使脚本更快，而且可能更健壮（不过需要正确地使用事务才能获得这样的好处）。

不幸的是，并非每种数据库都支持事务，因此当第一次打开连接时，PDO 需要在所谓的"自动提交"模式下运行。

自动提交模式意味着，如果数据库支持，运行的每个查询都有它自己的隐式事务，如果数据库不支持事务，则没有。

如果需要一个事务，则必须用 PDO::beginTransaction() 方法来启动。如果底层驱动不支持事务，则抛出一个 PDOException 异常（不管错误处理设置是怎样的，这都是一个严重的错误状态）。

一旦开始了事务，可用 PDO::commit() 或 PDO::rollBack()来完成，这取决于事务中的代码是否运行成功。

注意： PDO 仅在驱动层检查是否具有事务处理能力。如果某些运行时条件意味着事务不可用，且数据库服务接受请求去启动一个事务，PDO::beginTransaction() 将仍然返回 TRUE 而且没有错误。 试着在 MySQL 数据库的 MyISAM 数据表中使用事务就是一个很好的例子。

当脚本结束或连接即将被关闭时，如果尚有一个未完成的事务，那么 PDO 将自动回滚该事务。这种安全措施有助于在脚本意外终止时避免出现不一致的情况——如果没有显式地提交事务，那么假设是某个地方出错了，所以执行回滚来保证数据安全。

注意： 只有通过 PDO::beginTransaction() 启动一个事务后，才可能发生自动回滚。如果手动发出一条查询启动事务， 则 PDO 无法知晓，从而在必要时不能进行回滚。

在事务中执行批处理:

在下面例子中，假设为新员工创建一组条目，分配一个为23的ID。除了登记此人的基本数据之外，还需要记录他的工资。

两个更新分别完成起来很简单，但通过封闭在 PDO::beginTransaction() 和PDO::commit() 调用中，可以保证在更改完成之前，其他人无法看到这些更改。

如果发生了错误，catch 块回滚自事务启动以来发生的所有更改，并输出一条错误信息。

```
<?php
try {
  $dbh = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2',
      array(PDO::ATTR_PERSISTENT => true));
  echo "Connected\n";
} catch (Exception $e) {
  die("Unable to connect: " . $e->getMessage());
}

try {
  $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

  $dbh->beginTransaction();
  $dbh->exec("insert into staff (id, first, last) values (23, 'Joe', 'Bloggs')");
  $dbh->exec("insert into salarychange (id, amount, changedate)
      values (23, 50000, NOW())");
  $dbh->commit();

} catch (Exception $e) {
  $dbh->rollBack();
  echo "Failed: " . $e->getMessage();
}
?>
```

并不局限于在事务中更改，也可以发出复杂的查询来提取数据，还可以使用那些信息来构建更多的更改和查询；当事务激活时，可以保证其他人在操作进行当中无法作出更改。

PHP PDO 预处理语句与存储过程

很多更成熟的数据库都支持预处理语句的概念。

什么是预处理语句？可以把它看作是想要运行的 SQL 的一种编译过的模板，它可以使用变量参数进行定制。预处理语句可以带来两大好处：

- 查询仅需解析（或预处理）一次，但可以用相同或不同的参数执行多次。当查询准备好后，数据库将分析、编译和优化执行该查询的计划。对于复杂的查询，此过程要花费较长的时间，如果需要以不同参数多次重复相同的查询，那么该过程将大大降低应用程序的速度。通过使用预处理语句，可以避免重复分析/编译/优化周期。简言之，预处理语句占用更少的资源，因而运行得更快。
- 提供给预处理语句的参数不需要用引号括起来，驱动程序会自动处理。如果应用程序只使用预处理语句，可以确保不会发生SQL 注入。（然而，如果查询的其他部分是由未转义的输入来构建的，则仍存在 SQL 注入的风险）。

预处理语句如此有用，以至于它们唯一的特性是在驱动程序不支持的时PDO 将模拟处理。这样可以确保不管数据库是否具有这样的功能，都可以确保应用程序可以用相同的数据访问模式。

用预处理语句进行重复插入

下面例子通过用 name 和 value 替代相应的命名占位符来执行一个插入查询

```
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

// 插入一行
$name = 'one';
$value = 1;
$stmt->execute();

//  用不同的值插入另一行
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

用预处理语句进行重复插入

下面例子通过用 name 和 value 取代 ? 占位符的位置来执行一条插入查询。

```
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (?, ?)");
$stmt->bindParam(1, $name);
$stmt->bindParam(2, $value);

// 插入一行
$name = 'one';
$value = 1;
$stmt->execute();

// 用不同的值插入另一行
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

使用预处理语句获取数据

下面例子获取数据基于键值已提供的形式。用户的输入被自动用引号括起来，因此不会有 SQL 注入攻击的危险。

```
<?php
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where name = ?");
if ($stmt->execute(array($_GET['name']))) {
  while ($row = $stmt->fetch()) {
    print_r($row);
  }
}
?>
```

如果数据库驱动支持，应用程序还可以绑定输出和输入参数.输出参数通常用于从存储过程获取值。输出参数使用起来比输入参数要稍微复杂一些，因为当绑定一个输出参数时，必须知道给定参数的长度。如果为参数绑定的值大于建议的长度，就会产生一个错误。

带输出参数调用存储过程

```
<?php
$stmt = $dbh->prepare("CALL sp_returns_string(?)");
$stmt->bindParam(1, $return_value, PDO::PARAM_STR, 4000);

// 调用存储过程
$stmt->execute();

print "procedure returned $return_value\n";
?>
```

还可以指定同时具有输入和输出值的参数，其语法类似于输出参数。在下一个例子中，字符串"hello"被传递给存储过程，当存储过程返回时，hello 被替换为该存储过程返回的值。

带输入/输出参数调用存储过程

```
<?php
$stmt = $dbh->prepare("CALL sp_takes_string_returns_string(?)");
$value = 'hello';
$stmt->bindParam(1, $value, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 4000);

// 调用存储过程
$stmt->execute();

print "procedure returned $value\n";
?>
```

占位符的无效使用

```
<?php
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where name LIKE '%?%'");
$stmt->execute(array($_GET['name']));

// 占位符必须被用在整个值的位置
$stmt = $dbh->prepare("SELECT * FROM REGISTRY where name LIKE ?");
$stmt->execute(array("%$_GET[name]%"));
?>
```

PHP PDO 错误与错误处理

- **PDO::ERRMODE_SILENT**

  此为默认模式。 PDO 将只简单地设置错误码，可使用 PDO::errorCode() 和 PDO::errorInfo() 方法来检查语句和数据库对象。如果错误是由于对语句对象的调用而产生的，那么可以调用那个对象的 PDOStatement::errorCode() 或 PDOStatement::errorInfo() 方法。如果错误是由于调用数据库对象而产生的，那么可以在数据库对象上调用上述两个方法。

- **PDO::ERRMODE_WARNING**

  除设置错误码之外，PDO 还将发出一条传统的 E_WARNING 信息。如果只是想看看发生了什么问题且不中断应用程序的流程，那么此设置在调试/测试期间非常有用。

- **PDO::ERRMODE_EXCEPTION**

  除设置错误码之外，PDO 还将抛出一个 PDOException 异常类并设置它的属性来反射错误码和错误信息。此设置在调试期间也非常有用，因为它会有效地放大脚本中产生错误的点，从而可以非常快速地指出代码中有问题的潜在区域（记住：如果异常导致脚本终止，则事务被自动回滚）。

  异常模式另一个非常有用的是，相比传统 PHP 风格的警告，可以更清晰地构建自己的错误处理，而且比起静默模式和显式地检查每种数据库调用的返回值，异常模式需要的代码/嵌套更少。

创建 PDO 实例并设置错误模式

```
<?php
$dsn = 'mysql:dbname=testdb;host=127.0.0.1';
$user = 'dbuser';
$password = 'dbpass';

try {
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}

?>
```

注意： 不管当前是否设置了 PDO::ATTR_ERRMODE ， 如果连接失败，PDO::__construct() 将总是抛出一个 PDOException 异常。未捕获异常是致命的。

创建 PDO 实例并在构造函数中设置错误模式

```
<?php
$dsn = 'mysql:dbname=test;host=127.0.0.1';
$user = 'googleguy';
$password = 'googleguy';

/*
    使用 try/catch 围绕构造函数仍然有效，即使设置了 ERRMODE 为 WARNING，
    因为如果连接失败，PDO::__construct 将总是抛出一个  PDOException 异常。
*/
try {
    $dbh = new PDO($dsn, $user, $password, array(PDO::ATTR_ERRMODE => PDO::ERRMODE_WARNIN
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
    exit;
}

// 这里将导致 PDO 抛出一个 E_WARNING 级别的错误，而不是 一个异常 （当数据表不存在时）
$dbh->query("SELECT wrongcolumn FROM wrongtable");
?>
```

以上例程会输出：

```
Warning: PDO::query(): SQLSTATE[42S02]: Base table or view not found: 1146 Table 'test.wr
/tmp/pdo_test.php on line 18
add a note add a note
```

PHP PDO 大对象 (LOBs)

应用程序在某一时刻，可能需要在数据库中存储"大"数据。

"大"通常意味着"大约 4kb 或以上"，尽管某些数据库在数据达到"大"之前可以轻松地处理多达 32kb 的数据。大对象本质上可能是文本或二进制。

在 PDOStatement::bindParam() 或 PDOStatement::bindColumn()) 调用中使用 PDO::PARAM_LOB 类型码可以让 PDO 使用大数据类型。

PDO::PARAM_LOB 告诉 PDO 作为流来映射数据，以便能使用 PHP Streams API 来操作。

从数据库中显示一张图片

下面例子绑定一个 LOB 到 $lob 变量，然后用 fpassthru() 将其发送到浏览器。因为 LOB 代表一个流，所以类似 fgets()、fread() 以及 stream_get_contents() 这样的函数都可以用在它上面。

```
<?php
$db = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$stmt = $db->prepare("select contenttype, imagedata from images where id=?");
$stmt->execute(array($_GET['id']));
$stmt->bindColumn(1, $type, PDO::PARAM_STR, 256);
$stmt->bindColumn(2, $lob, PDO::PARAM_LOB);
$stmt->fetch(PDO::FETCH_BOUND);

header("Content-Type: $type");
fpassthru($lob);
?>
```

插入一张图片到数据库

下面例子打开一个文件并将文件句柄传给 PDO 来做为一个 LOB 插入。PDO尽可能地让数据库以最有效的方式获取文件内容。

```
<?php
$db = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$stmt = $db->prepare("insert into images (id, contenttype, imagedata) values (?, ?, ?)");
$id = get_new_id(); // 调用某个函数来分配一个新 ID

// 假设处理一个文件上传
// 可以在 PHP 文档中找到更多的信息

$fp = fopen($_FILES['file']['tmp_name'], 'rb');

$stmt->bindParam(1, $id);
$stmt->bindParam(2, $_FILES['file']['type']);
$stmt->bindParam(3, $fp, PDO::PARAM_LOB);

$db->beginTransaction();
$stmt->execute();
$db->commit();
?>
```

插入一张图片到数据库：Oracle

对于从文件插入一个 lob，Oracle略有不同。必须在事务之后进行插入，否则当执行查询时导致新近插入 LOB 将以0长度被隐式提交：

```
<?php
$db = new PDO('oci:', 'scott', 'tiger');
$stmt = $db->prepare("insert into images (id, contenttype, imagedata) " .
"VALUES (?, ?, EMPTY_BLOB()) RETURNING imagedata INTO ?");
$id = get_new_id(); // 调用某个函数来分配一个新 ID

// 假设处理一个文件上传
// 可以在 PHP 文档中找到更多的信息

$fp = fopen($_FILES['file']['tmp_name'], 'rb');

$stmt->bindParam(1, $id);
$stmt->bindParam(2, $_FILES['file']['type']);
$stmt->bindParam(3, $fp, PDO::PARAM_LOB);

$stmt->beginTransaction();
$stmt->execute();
$stmt->commit();
?>
```

PDO::beginTransaction

PDO::beginTransaction 启动一个事务(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDO::beginTransaction ( void )
```

关闭自动提交模式。自动提交模式被关闭的同时，通过 PDO 对象实例对数据库做出的更改直到调用 PDO::commit() 结束事务才被提交。

调用 PDO::rollBack() 将回滚对数据库做出的更改并将数据库连接返回到自动提交模式。

包括 MySQL 在内的一些数据库，当发出一条类似 DROP TABLE 或 CREATE TABLE 这样的 DDL 语句时，会自动进行一个隐式地事务提交。

隐式地提交将阻止你在此事务范围内回滚任何其他更改。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

回滚一个事务

下面例子在回滚此更改前开始一个事务并发出两条修改数据库的语句。

但在 MySQL 中，DROP TABLE 语句自动提交事务，使得在此事务中的任何更改都不会被回滚。

```
<?php
/* 开始一个事务，关闭自动提交 */
$dbh->beginTransaction();

/*  更改数据库架构及数据 */
$sth = $dbh->exec("DROP TABLE fruit");
$sth = $dbh->exec("UPDATE dessert
    SET name = 'hamburger'");

/*  识别出错误并回滚更改 */
$dbh->rollBack();

/* 数据库连接现在返回到自动提交模式 */
?>
```

PDO::commit

PDO::commit提交一个事务(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDO::commit ( void )
```

提交一个事务，数据库连接返回到自动提交模式直到下次调用 PDO::beginTransaction() 开始一个新的事务为止。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

提交一个基础事务

```
<?php
/* 开始一个事务，关闭自动提交 */
$dbh->beginTransaction();

/* 在全有或全无的基础上插入多行记录（要么全部插入，要么全部不插入） */
$sql = 'INSERT INTO fruit
    (name, colour, calories)
    VALUES (?, ?, ?)';

$sth = $dbh->prepare($sql);

foreach ($fruits as $fruit) {
    $sth->execute(array(
        $fruit->name,
        $fruit->colour,
        $fruit->calories,
    ));
}

/* 提交更改 */
$dbh->commit();

/* 现在数据库连接返回到自动提交模式 */
?>
```

提交一个DDL事务

```
<?php
/*  开始一个事务，关闭自动提交 */
$dbh->beginTransaction();

/* Change the database schema */
$sth = $dbh->exec("DROP TABLE fruit");

/* 更改数据库架构 */
$dbh->commit();

/* 现在数据库连接返回到自动提交模式 */
?>
```

注意：并不是所有数据库都允许使用DDL语句进行事务操作：有些会产生错误，而其他一些（包括MySQL）会在遇到第一个DDL语句后就自动提交事务。

PDO::__construct

PDO::__construct — 创建一个表示数据库连接的 PDO 实例(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
PDO::__construct ( string $dsn [, string $username [, string $password [, array $driver_o
```

创建一个表示连接到请求数据库的数据库连接 PDO 实例。

参数说明

- **dsn**：数据源名称或叫做 DSN，包含了请求连接到数据库的信息。
- **username**： DSN字符串中的用户名。对于某些PDO驱动，此参数为可选项。
- **password**： DSN字符串中的密码。对于某些PDO驱动，此参数为可选项。
- **driver_options**： 一个具体驱动的连接选项的键=>值数组。

返回值

成功则返回一个PDO对象。

错误／异常

如果试图连接到请求的数据库失败，则PDO::__construct() 抛出一个 PDO异常（PDOException）。

实例

通过调用驱动程序创建一个PDO实例

```
<?php
/* 通过调用驱动程序创建一个PDO实例 */
$dsn = 'mysql:dbname=testdb;host=127.0.0.1';
$user = 'dbuser';
$password = 'dbpass';

try {
    $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}

?>
```

PDO::errorCode

PDO::errorCode — 获取跟数据库句柄上一次操作相关的 SQLSTATE(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
mixed PDO::errorCode ( void )
```

返回值

返回一个 SQLSTATE，一个由5个字母或数字组成的在 ANSI SQL 标准中定义的标识符。 简要地说，一个 SQLSTATE 由前面两个字符的类值和后面三个字符的子类值组成。

如果数据库句柄没有进行操作，则返回 NULL 。

实例

取得一个 SQLSTATE 码

```
/* 引发一个错误 -- BONES 数据表不存在 */
$dbh->exec("INSERT INTO bones(skull) VALUES ('lucy')");

echo "\nPDO::errorCode(): ";
print $dbh->errorCode();
?>
```

以上例程会输出：

```
PDO::errorCode(): 42S02
```

PDO::errorInfo

PDO::errorCode — 返回最后一次操作数据库的错误信息(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
public array PDO::errorInfo ( void )
```

返回值

返回一个数组，该数组包含了最后一次操作数据库的错误信息描述。

数组内容如下：

元素	信息
0	SQLSTATE 错误码 (5个字母或数字组成的在 ANSI SQL 标准中定义的标识符).
1	错误代码
2	错误信息

注意：如果数据库句柄没有进行操作，则返回 NULL 。

实例

显示errorInfo()中关于PDO_ODBC连接到DB2数据库的错误信息

```
<?php
/* 错误的SQL语法 */
$stmt = $dbh->prepare('bogus sql');
if (!$stmt) {
    echo "\nPDO::errorInfo():\n";
    print_r($dbh->errorInfo());
}
?>
```

以上例程会输出：

```
PDO::errorInfo():
Array
(
    [0] => HY000
    [1] => 1
    [2] => near "bogus": syntax error
)
```

PDO::exec

PDO::exec — 执行一条 SQL 语句，并返回受影响的行数(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
int PDO::exec ( string $statement )
```

PDO::exec() 在一个单独的函数调用中执行一条 SQL 语句，返回受此语句影响的行数。

PDO::exec() 不会从一条 SELECT 语句中返回结果。对于在程序中只需要发出一次的 SELECT 语句，可以考虑使用 PDO::query()。

参数说明：

statement： 要被预处理和执行的 SQL 语句。

返回值

PDO::exec() 返回受修改或删除 SQL 语句影响的行数。如果没有受影响的行，则 PDO::exec() 返回 0。

下面例子依赖 PDO::exec() 的返回值是不正确的，其中受影响行数为 0 的语句会导致调用 die()：

```
<?php
$db->exec() or die(print_r($db->errorInfo(), true));
?>
```

实例

执行一条 **DELETE** 语句

计算由一条不带 WHERE 字句的 DELETE 语句删除的行数。

```
<?php
$dbh = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');

/*  删除 FRUIT 数据表中满足条件的所有行 */
$count = $dbh->exec("DELETE FROM fruit WHERE colour = 'red'");

/* 返回被删除的行数 */
print("Deleted $count rows.\n");
?>
```

以上例程会输出：

```
Deleted 1 rows.
```

PDO::getAttribute

PDO::getAttribute — 取回一个数据库连接的属性(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
mixed PDO::getAttribute ( int $attribute )
```

此函数（方法）返回一个数据库连接的属性值。 取回 PDOStatement 属性，请参阅 PDOStatement::getAttribute()。

注意有些数据库/驱动可能不支持所有的数据库连接属性。

参数说明：

attribute： PDO::ATTR_* 常量中的一个。下列为应用到数据库连接中的常量：

- *PDO::ATTR_AUTOCOMMIT*
- *PDO::ATTR_CASE*
- *PDO::ATTR_CLIENT_VERSION*
- *PDO::ATTR_CONNECTION_STATUS*
- *PDO::ATTR_DRIVER_NAME*
- *PDO::ATTR_ERRMODE*
- *PDO::ATTR_ORACLE_NULLS*
- *PDO::ATTR_PERSISTENT*
- *PDO::ATTR_PREFETCH*
- *PDO::ATTR_SERVER_INFO*
- *PDO::ATTR_SERVER_VERSION*
- *PDO::ATTR_TIMEOUT*

返回值

成功调用则返回请求的 PDO 属性值。不成功则返回 null。

实例

取回数据库连接属性

```
<?php
$conn = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');
$attributes = array(
    "AUTOCOMMIT", "ERRMODE", "CASE", "CLIENT_VERSION", "CONNECTION_STATUS",
    "ORACLE_NULLS", "PERSISTENT", "PREFETCH", "SERVER_INFO", "SERVER_VERSION",
    "TIMEOUT"
);

foreach ($attributes as $val) {
    echo "PDO::ATTR_$val: ";
    echo $conn->getAttribute(constant("PDO::ATTR_$val")) . "\n";
}
?>
```

PDO::getAvailableDrivers

PDO::getAvailableDrivers — 返回一个可用驱动的数组(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
static array PDO::getAvailableDrivers ( void )
```

```
array pdo_drivers ( void )
```

此函数（方法）返回所有当前可用在 PDO::__construct() 的参数 DSN 中的 PDO 驱动。

返回值

PDO::getAvailableDrivers() 返回一个 包含可用 PDO 驱动名字的数组。如果没有可用的驱动，则返回一个空数组。

实例

一个 PDO::getAvailableDrivers() 的例子

```
<?php
print_r(PDO::getAvailableDrivers());
?>
```

以上例程的输出类似于：

```
Array
(
    [0] => mysql
    [1] => sqlite
)
```

PDO::inTransaction

PDO::inTransaction — 检查是否在一个事务内(PHP 5 >= 5.3.3, Bundled pdo_pgsql)

说明

语法

```
bool PDO::inTransaction ( void )
```

检查驱动内的一个事务当前是否处于激活。此方法仅对支持事务的数据库驱动起作用。

参数

此函数没有参数。

返回值

如果当前事务处于激活，则返回 TRUE， 否则返回 FALSE 。

PDO::lastInsertId

PDO::lastInsertId — 返回最后插入行的ID或序列值(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
string PDO::lastInsertId ([ string $name = NULL ] )
```

返回最后插入行的ID，或者是一个序列对象最后的值，取决于底层的驱动。比如，PDO_PGSQL() 要求为 name 参数指定序列对象的名称。

注意： 在不同的 PDO 驱动之间，此方法可能不会返回一个有意义或一致的结果，因为底层数据库可能不支持自增字段或序列的概念。

参数

name 应该返回ID的那个序列对象的名称。

返回值

如果没有为参数 name 指定序列名称，PDO::lastInsertId() 则返回一个表示最后插入数据库那一行的行ID的字符串。

如果为参数 name 指定了序列名称，PDO::lastInsertId() 则返回一个表示从指定序列对象取回最后的值的字符串。

如果当前 PDO 驱动不支持此功能，则 PDO::lastInsertId() 触发一个 IM001 SQLSTATE 。

PDO::prepare

PDO::prepare — 准备要执行的SQL语句并返回一个 PDOStatement 对象(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
public PDOStatement PDO::prepare ( string $statement [, array $driver_options = array() ]
```

为 PDOStatement::execute() 方法准备要执行的SQL语句，SQL语句可以包含零个或多个命名（:name）或问号（?）参数标记，参数在SQL执行时会被替换。

你不能在 SQL 语句中同时包含命名（:name）或问号（?）参数标记，只能选择其中一种风格。

预处理 SQL 语句中的参数在使用PDOStatement::execute()方法时会传递真实的参数。

参数

statement 合法的SQL语句。

driver_options 此数组包含一个或多个 key=>value 对来设置 PDOStatement 对象的属性，最常使用到是将PDO::ATTR_CURSOR值设置为PDO::CURSOR_SCROLL来请求一个可滚动游标。

返回值

如果成功，PDO::prepare()返回PDOStatement对象，如果失败返回 FALSE 或抛出异常 PDOException 。

实例

使用命名（:name）参数来准备SQL语句

```
<?php
/* 通过数组值向预处理语句传递值 */
$sql = 'SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour';
$sth = $dbh->prepare($sql, array(PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY));
$sth->execute(array(':calories' => 150, ':colour' => 'red'));
$red = $sth->fetchAll();
$sth->execute(array(':calories' => 175, ':colour' => 'yellow'));
$yellow = $sth->fetchAll();
?>
```

使用问号（?）参数来准备SQL语句

```
<?php
/* 通过数组值向预处理语句传递值 */
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->execute(array(150, 'red'));
$red = $sth->fetchAll();
$sth->execute(array(175, 'yellow'));
$yellow = $sth->fetchAll();
?>
```

PDO::query

PDO::query — 执行 SQL 语句，返回PDOStatement对象,可以理解为结果集(PHP 5 >= 5.1.0, PECL pdo >= 0.2.0)

说明

语法

```
public PDOStatement PDO::query ( string $statement )
```

```
public PDOStatement PDO::query ( string $statement , int $PDO::FETCH_COLUMN , int $colno
```

```
public PDOStatement PDO::query ( string $statement , int $PDO::FETCH_CLASS , string $clas
```

```
public PDOStatement PDO::query ( string $statement , int $PDO::FETCH_INTO , object $objec
```

PDO::query() 在一个单独的函数中调用并执行 SQL 语句, 返回结果集 (如果有),语句作为一个PDOStatement对象返回。

参数

statement 要执行的SQL语句。

返回值

如果成功，PDO::query()返回PDOStatement对象，如果失败返回 FALSE 。

实例

PDO::query实例

遍历输出结果集：

```
<?php
function getFruit($conn) {
    $sql = 'SELECT name, color, calories FROM fruit ORDER BY name';
    foreach ($conn->query($sql) as $row) {
        print $row['name'] . "\t";
        print $row['color'] . "\t";
        print $row['calories'] . "\n";
    }
}
?>
```

以上输出结果为：

```
apple   red     150
banana  yellow  250
kiwi    brown   75
lemon   yellow  25
orange  orange  300
pear    green   150
watermelon      pink    90
```

PDO::quote

PDO::quote — 为SQL语句中的字符串添加引号。(PHP 5 >= 5.1.0, PECL pdo >= 0.2.1)

说明

语法

```
public string PDO::quote ( string $string [, int $parameter_type = PDO::PARAM_STR ] )
```

PDO::quote() 为SQL语句中的字符串添加引号或者转义特殊字符串。

参数

string 要添加引号的字符串。

parameter_type 为驱动程序提供数据类型。

返回值

返回一个带引号的字符串，理论上可以安全的传递到SQL语句中并执行。如果该驱动程序不支持则返回FALSE。

实例

为普通字符串添加引号

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Simple string */
$string = 'Nice';
print "Unquoted string: $string\n";
print "Quoted string: " . $conn->quote($string) . "\n";
?>
```

以上输出结果为：

```
Unquoted string: Nice
Quoted string: 'Nice'
```

转义特殊字符串

```
<?php
$conn = new PDO('sqlite:/home/lynn/music.sql3');

/* Dangerous string */
$string = 'Naughty \' string';
print "Unquoted string: $string\n";
print "Quoted string:" . $conn->quote($string) . "\n";
?>
```

以上例程会输出：

```
Unquoted string: Naughty ' string
Quoted string: 'Naughty '' string'
```

PDO::rollBack

PDO::rollBack — 回滚一个事务(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDO::rollBack ( void )
```

回滚由 PDO::beginTransaction() 发起的当前事务。如果没有事务激活，将抛出一个 PDOException 异常。

如果数据库被设置成自动提交模式，此函数（方法）在回滚事务之后将恢复自动提交模式。

包括 MySQL 在内的一些数据库， 当在一个事务内有类似删除或创建数据表等 DLL 语句时，会自动导致一个隐式地提交。隐式地提交将无法回滚此事务范围内的任何更改。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

回滚一个事务

下面例子在回滚更改之前开始一个事务并发出两条修改数据库的语句。但在 MySQL 中，DROP TABLE 语句自动提交事务，因此在此事务内的任何更改都不会被回滚。

```
<?php
/* 开始一个事务，关闭自动提交 */
$dbh->beginTransaction();

/* 更改数据库架构和数据  */
$sth = $dbh->exec("DROP TABLE fruit");
$sth = $dbh->exec("UPDATE dessert
    SET name = 'hamburger'");

/*  识别错误且回滚更改  */
$dbh->rollBack();

/*  此时数据库连接恢复到自动提交模式  */
?>
```

PDO::setAttribute

PDO::setAttribute — 设置属性(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDO::setAttribute ( int $attribute , mixed $value )
```

设置数据库句柄属性。下面列出了一些可用的通用属性；有些驱动可能使用另外的特定属性。

- *PDO::ATTR_CASE*：强制列名为指定的大小写。
    - *PDO::CASE_LOWER*：强制列名小写。
    - *PDO::CASE_NATURAL*：保留数据库驱动返回的列名。
    - *PDO::CASE_UPPER*：强制列名大写。
- *PDO::ATTR_ERRMODE*：错误报告。
    - *PDO::ERRMODE_SILENT*： 仅设置错误代码。
    - *PDO::ERRMODE_WARNING*: 引发 *E_WARNING* 错误
    - *PDO::ERRMODE_EXCEPTION*: 抛出 *exceptions* 异常。
- *PDO::ATTR_ORACLE_NULLS* （在所有驱动中都可用，不仅限于Oracle）： 转换 NULL 和空字符串。
    - *PDO::NULL_NATURAL*: 不转换。
    - *PDO::NULL_EMPTY_STRING*： 将空字符串转换成 `NULL` 。
    - *PDO::NULL_TO_STRING*: 将 NULL 转换成空字符串。
- *PDO::ATTR_STRINGIFY_FETCHES*: 提取的时候将数值转换为字符串。 需要 *bool*。
- *PDO::ATTR_STATEMENT_CLASS*： 设置从PDOStatement派生的用户提供的语句类。 不能用于持久的PDO实例。 需要 *array(string 类名, array(mixed 构造函数的参数))*。

- *PDO::ATTR_TIMEOUT*： 指定超时的秒数。并非所有驱动都支持此选项，这意味着驱动和驱动之间可能会有差异。比如，SQLite等待的时间达到此值后就放弃获取可写锁，但其他驱动可能会将此值解释为一个连接或读取超时的间隔。 需要 int 类型。
- *PDO::ATTR_AUTOCOMMIT* （在OCI，Firebird 以及 MySQL中可用）： 是否自动提交每个单独的语句。
- *PDO::ATTR_EMULATE_PREPARES* 启用或禁用预处理语句的模拟。 有些驱动不支持或有限度地支持本地预处理。使用此设置强制PDO总是模拟预处理语句（如果为 `TRUE` ），或试着使用本地预处理语句（如果为 `FALSE` ）。如果驱动不能成功预处理当前查询，它将总是回到模拟预处理语句上。 需要 *bool* 类型。
- *PDO::MYSQL_ATTR_USE_BUFFERED_QUERY* （在MySQL中可用）： 使用缓冲查询。
- *PDO::ATTR_DEFAULT_FETCH_MODE*： 设置默认的提取模式。关于模式的说明可以在 *PDOStatement::fetch()* 文档找到。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

PDOStatement::bindColumn

PDOStatement::bindColumn — 绑定一列到一个 PHP 变量(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDOStatement::bindColumn ( mixed $column , mixed &$param [, int $type [, int $maxlen
```

安排一个特定的变量绑定到一个查询结果集中给定的列。每次调用 PDOStatement::fetch() 或 PDOStatement::fetchAll() 都将更新所有绑定到列的变量。

注意：在语句执行前 PDO 有关列的信息并非总是可用，可移植的应用应在 PDOStatement::execute() 之后 调用此函数（方法）。 但是，当使用 PgSQL 驱动 时，要想能绑定一个 LOB 列作为流，应用程序必须在调用 PDOStatement::execute() 之前 调用此方法，否则大对象 OID 作为一个整数返回。

参数

column 结果集中的列号（从1开始索引）或列名。如果使用列名，注意名称应该与由驱动返回的列名大小写保持一致。

param 将绑定到列的 PHP 变量名称

type 通过 PDO::PARAM_* 常量指定的参数的数据类型。

maxlen 预分配提示。

driverdata 驱动的可选参数。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

把结果集输出绑定到 PHP 变量

绑定结果集中的列到PHP变量是一种使每行包含的数据在应用程序中立即可用的有效方法。下面的例子演示了 PDO 怎样用多种选项和缺省值绑定和检索列。

```
<?php
function readData($dbh) {
  $sql = 'SELECT name, colour, calories FROM fruit';
  try {
    $stmt = $dbh->prepare($sql);
    $stmt->execute();

    /*  通过列号绑定  */
    $stmt->bindColumn(1, $name);
    $stmt->bindColumn(2, $colour);

    /*  通过列名绑定  */
    $stmt->bindColumn('calories', $cals);

    while ($row = $stmt->fetch(PDO::FETCH_BOUND)) {
      $data = $name . "\t" . $colour . "\t" . $cals . "\n";
      print $data;
    }
  }
  catch (PDOException $e) {
    print $e->getMessage();
  }
}
readData($dbh);
?>
```

以上例程会输出：

```
apple   red     150
banana  yellow  175
kiwi    green   75
orange  orange  150
mango   red     200
strawberry      red     25
```

PDOStatement::bindParam

PDOStatement::bindParam — 绑定一个参数到指定的变量名(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDOStatement::bindParam ( mixed $parameter , mixed &$variable [, int $data_type = PD
```

绑定一个PHP变量到用作预处理的SQL语句中的对应命名占位符或问号占位符。 不同于 PDOStatement::bindValue()，此变量作为引用被绑定，并只在 PDOStatement::execute() 被调用的时候才取其值。

大多数参数是输入参数，即，参数以只读的方式用来建立查询。一些驱动支持调用存储过程并作为输出参数返回数据，一些支持作为输入/输出参数，既发送数据又接收更新后的数据。

参数

parameter 参数标识符。对于使用命名占位符的预处理语句，应是类似 :name 形式的参数名。对于使用问号占位符的预处理语句，应是以1开始索引的参数位置。

variable 绑定到 SQL 语句参数的 PHP 变量名。

data_type 使用 PDO::PARAM_* 常量明确地指定参数的类型。要从一个存储过程中返回一个 INOUT 参数，需要为 data_type 参数使用按位或操作符去设置 PDO::PARAM_INPUT_OUTPUT 位。

length 预分配提示。

driverdata 数据类型的长度。为表明参数是一个存储过程的 OUT 参数，必须明确地设置此长度。

driver_options

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

执行一条使用命名占位符的预处理语句

```
<?php
/* 通过绑定的 PHP 变量执行一条预处理语句  */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->bindParam(':calories', $calories, PDO::PARAM_INT);
$sth->bindParam(':colour', $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

执行一条使用问号占位符的预处理语句

```
<?php
/*  通过绑定的 PHP 变量执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories, PDO::PARAM_INT);
$sth->bindParam(2, $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

使用 **INOUT** 参数调用一个存储过程

```
<?php
/* 使用 INOUT 参数调用一个存储过程 */
$colour = 'red';
$sth = $dbh->prepare('CALL puree_fruit(?)');
$sth->bindParam(1, $colour, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 12);
$sth->execute();
print("After pureeing fruit, the colour is: $colour");
?>
```

PDOStatement::bindValue

PDOStatement::bindValue — 把一个值绑定到一个参数(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDOStatement::bindValue ( mixed $parameter , mixed $value [, int $data_type = PDO::P
```

绑定一个值到用作预处理的 SQL 语句中的对应命名占位符或问号占位符。

参数

parameter 参数标识符。对于使用命名占位符的预处理语句，应是类似 :name 形式的参数名。对于使用问号占位符的预处理语句，应是以1开始索引的参数位置。

value 绑定到参数的值

data_type 使用 PDO::PARAM_* 常量明确地指定参数的类型。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

执行一条使用命名占位符的预处理语句

```
<?php
/* 通过绑定的 PHP 变量执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->bindValue(':calories', $calories, PDO::PARAM_INT);
$sth->bindValue(':colour', $colour, PDO::PARAM_STR);
$sth->execute();
?>
```

执行一条使用问号占位符的预处理语句

```
<?php
/* 通过绑定的 PHP 变量执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindValue(1, $calories, PDO::PARAM_INT);
$sth->bindValue(2, $colour, PDO::PARAM_STR);
$sth->execute();
?>
```

PDOStatement::closeCursor

PDOStatement::closeCursor — 关闭游标，使语句能再次被执行。(PHP 5 >= 5.1.0, PECL pdo >= 0.9.0)

说明

语法

```
bool PDOStatement::closeCursor ( void )
```

PDOStatement::closeCursor() 释放到数据库服务的连接，以便发出其他 SQL 语句，但使语句处于一个可以被再次执行的状态。

当上一个执行的 PDOStatement 对象仍有未取行时，此方法对那些不支持再执行一个 PDOStatement 对象的数据库驱动非常有用。 如果数据库驱动受此限制，则可能出现失序错误的问题。

PDOStatement::closeCursor() 要么是一个可选驱动的特有方法（效率最高）来实现，要么是在没有驱动特定的功能时作为一般的PDO 备用来实现。一般的备用语义上与下面的 PHP 代码相同：

```
<?php
do {
    while ($stmt->fetch())
        ;
    if (!$stmt->nextRowset())
        break;
} while (true);
?>
```

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

一个 PDOStatement::closeCursor() 的例子

在下面例子中，$stmt PDOStatement 对象返回多行，但应用程序只取第一行，让 PDOStatement 对象处于一个有未取行的状态。为确保应用程序对所有数据库驱动都能正常运行，在执行 $otherStmt PDOStatement 对象前，$stmt 调用一次 PDOStatement::closeCursor() 。

```
<?php
/* 创建一个 PDOStatement 对象 */
$stmt = $dbh->prepare('SELECT foo FROM bar');

/* 创建第二个 PDOStatement 对象 */
$otherStmt = $dbh->prepare('SELECT foobaz FROM foobar');

/* 执行第一条语句 */
$stmt->execute();

/*  从结果集中只取出第一行 */
$stmt->fetch();

/* The following call to closeCursor() may be required by some drivers */
$stmt->closeCursor();

/*  现在可以执行第二条语句了 */
$otherStmt->execute();
?>
```

PDOStatement::columnCount

PDOStatement::columnCount — 返回结果集中的列数。(PHP 5 >= 5.1.0, PECL pdo >= 0.2.0)

说明

语法

```
int PDOStatement::columnCount ( void )
```

使用 PDOStatement::columnCount() 返回由 PDOStatement 对象代表的结果集中的列数。

如果是由 PDO::query() 返回的 PDOStatement 对象，则列数计算立即可用。

如果是由 PDO::prepare() 返回的 PDOStatement 对象，则在调用 PDOStatement::execute() 之前都不能准确地计算出列数。

返回值

返回由 PDOStatement 对象代表的结果集中的列数。如果没有结果集，则 PDOStatement::columnCount() 返回 0。

实例

计算列数

下面例子演示如何使用 PDOStatement::columnCount() 操作一个结果集和一个空集。

```
<?php
$dbh = new PDO('odbc:sample', 'db2inst1', 'ibmdb2');

$sth = $dbh->prepare("SELECT name, colour FROM fruit");

/*  计算一个（不存在）的结果集中的列数 */
$colcount = $sth->columnCount();
print("Before execute(), result set has $colcount columns (should be 0)\n");

$sth->execute();

/* 计算结果集中的列数 */
$colcount = $sth->columnCount();
print("After execute(), result set has $colcount columns (should be 2)\n");

?>
```

以上例程会输出：

```
Before execute(), result set has 0 columns (should be 0)
After execute(), result set has 2 columns (should be 2)
```

PDOStatement::debugDumpParams

PDOStatement::debugDumpParams — 打印一条 SQL 预处理命令(PHP 5 >= 5.1.0, PECL pdo >= 0.9.0)

说明

语法

```
bool PDOStatement::debugDumpParams ( void )
```

直接打印出一条预处理语句包含的信息。提供正在使用的 SQL 查询、所用参数（Params）的数目、参数的清单、参数名、用一个整数表示的参数类型（paramtype）、键名或位置、值、以及在查询中的位置（如果当前 POD 驱动不支持，则为-1）。

此为一个用于调试的功能，在正常输出的情况下直接输出数据。

提示： 和直接将结果输出到浏览器一样，可使用输出控制函数来捕获当前函数的输出，然后(例如)保存到一个 string 中。

只打印此时此刻语句中的参数。额外的参数不存储在语句中，也就不会被输出。

返回值

没有返回值。

实例

PDOStatement::debugDumpParams() 使用命名参数的例子

```
<?php
/* 通过绑定 PHP 变量执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->bindParam(':calories', $calories, PDO::PARAM_INT);
$sth->bindValue(':colour', $colour, PDO::PARAM_STR, 12);
$sth->execute();

$sth->debugDumpParams();

?>
```

以上例程会输出：

```
SQL: [96] SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour
Params:  2
Key: Name: [9] :calories
paramno=-1
name=[9] ":calories"
is_param=1
param_type=1
Key: Name: [7] :colour
paramno=-1
name=[7] ":colour"
is_param=1
param_type=2
```

PDOStatement::debugDumpParams() 使用未命名参数的例子

```
<?php

/* 通过绑定 PHP 变量执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$name = 'apple';

$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories, PDO::PARAM_INT);
$sth->bindValue(2, $colour, PDO::PARAM_STR);
$sth->execute();

$sth->debugDumpParams();

?>
```

以上例程会输出：

```
SQL: [82] SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?
Params:  2
Key: Position #0:
paramno=0
name=[0] ""
is_param=1
param_type=1
Key: Position #1:
paramno=1
name=[0] ""
is_param=1
param_type=2
```

PDOStatement::errorCode

PDOStatement::errorCode — 获取跟上一次语句句柄操作相关的 SQLSTATE(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
string PDOStatement::errorCode ( void )
```

与 PDO::errorCode() 相同，只是 PDOStatement::errorCode() 只取回 PDOStatement 对象执行操作中的错误码。

返回值

没有返回值。

实例

取回一个 SQLSTATE 码

```
<?php
/* 引发一个错误 --  BONES 数据表不存在 */
$err = $dbh->prepare('SELECT skull FROM bones');
$err->execute();

echo "\nPDOStatement::errorCode(): ";
print $err->errorCode();
?>
```

以上例程会输出：

```
PDOStatement::errorCode(): 42S02
```

PDOStatement::errorInfo

PDOStatement::errorInfo — 获取跟上一次语句句柄操作相关的扩展错误信息(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
array PDOStatement::errorInfo ( void )
```

PDOStatement::errorInfo() 返回一个关于上一次语句句柄执行操作的错误信息的数组 。该数组包含下列字段：

元素	信息
0	SQLSTATE 错误码（一个由5个字母或数字组成的在 ANSI SQL 标准中定义的标识符）。
1	具体驱动错误码。
2	具体驱动错误信息。

实例

显示连接到DB2数据库的 **PDO_ODBC** 连接的 **errorInfo()** 的字段

```
<?php
/* 激发一个错误 --  BONES 数据表不存在 */
$sth = $dbh->prepare('SELECT skull FROM bones');
$sth->execute();

echo "\nPDOStatement::errorInfo():\n";
$arr = $sth->errorInfo();
print_r($arr);
?>
<pre>
PDOStatement::errorCode(): 42S02
```

以上例程会输出：

```
PDOStatement::errorInfo():
Array
(
    [0] => 42S02
    [1] => -204
    [2] => [IBM][CLI Driver][DB2/LINUX] SQL0204N  "DANIELS.BONES" is an undefined name.
)
```

PDOStatement::execute

PDOStatement::execute — 执行一条预处理语句(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
bool PDOStatement::execute ([ array $input_parameters ] )
```

执行预处理过的语句。如果预处理过的语句含有参数标记，必须选择下面其中一种做法：

- 调用 PDOStatement::bindParam() 绑定 PHP 变量到参数标记：如果有的话，通过关联参数标记绑定的变量来传递输入值和取得输出值
- 或传递一个只作为输入参数值的数组

参数

input_parameters

一个元素个数和将被执行的 SQL 语句中绑定的参数一样多的数组。所有的值作为 PDO::PARAM_STR 对待。

不能绑定多个值到一个单独的参数；比如，不能绑定两个值到 IN（）子句中一个单独的命名参数。

绑定的值不能超过指定的个数。如果在 input_parameters 中存在比 PDO::prepare() 预处理的 SQL 指定的多的键名，则此语句会失败并发出一个错误。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

执行一条绑定变量的预处理语句

```
<?php
/*  通过绑定 PHP 变量执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->bindParam(':calories', $calories, PDO::PARAM_INT);
$sth->bindParam(':colour', $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

使用一个含有插入值的数组执行一条预处理语句（命名参数）

```
<?php
/* 通过传递一个含有插入值的数组执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour');
$sth->execute(array(':calories' => $calories, ':colour' => $colour));
?>
```

使用一个含有插入值的数组执行一条预处理语句（占位符）

```
<?php
/*  通过传递一个插入值的数组执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->execute(array($calories, $colour));
?>
```

执行一条问号占位符的预处理语句

```
<?php
/* 通过绑定 PHP 变量执行一条预处理语句 */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories, PDO::PARAM_INT);
$sth->bindParam(2, $colour, PDO::PARAM_STR, 12);
$sth->execute();
?>
```

使用数组执行一条含有 IN 子句的预处理语句

```
<?php
/*  使用一个数组的值执行一条含有 IN 子句的预处理语句 */
$params = array(1, 21, 63, 171);
/*  创建一个填充了和params相同数量占位符的字符串 */
$place_holders = implode(',', array_fill(0, count($params), '?'));

/*
    对于 $params 数组中的每个值，要预处理的语句包含足够的未命名占位符 。
    语句被执行时， $params 数组中的值被绑定到预处理语句中的占位符。
    这和使用 PDOStatement::bindParam() 不一样，因为它需要一个引用变量。
    PDOStatement::execute() 仅作为通过值绑定的替代。
*/
$sth = $dbh->prepare("SELECT id, name FROM contacts WHERE id IN ($place_holders)");
$sth->execute($params);
?>
```

PDOStatement::fetch

PDOStatement::fetch — 从结果集中获取下一行(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
mixed PDOStatement::fetch ([ int $fetch_style [, int $cursor_orientation = PDO::FETCH_ORI
```

从一个 PDOStatement 对象相关的结果集中获取下一行。fetch_style 参数决定 POD 如何返回行。

参数

fetch_style

控制下一行如何返回给调用者。此值必须是 PDO::FETCH_* 系列常量中的一个，缺省为 PDO::ATTR_DEFAULT_FETCH_MODE 的值 （默认为 PDO::FETCH_BOTH ）。

- *PDO::FETCH_ASSOC*：返回一个索引为结果集列名的数组
- *PDO::FETCH_BOTH*（默认）：返回一个索引为结果集列名和以0开始的列号的数组
- *PDO::FETCH_BOUND*：返回 `TRUE` ，并分配结果集中的列值给 PDOStatement::bindColumn() 方法绑定的 PHP 变量。
- *PDO::FETCH_CLASS*：返回一个请求类的新实例，映射结果集中的列名到类中对应的属性名。如果 `fetch_style` 包含 PDO::FETCH*CLASSTYPE*（例如：_*PDO::FETCH_CLASS* | *PDO::FETCH_CLASSTYPE*），则类名由第一列的值决定
- *PDO::FETCH_INTO*：更新一个被请求类已存在的实例，映射结果集中的列到类中命名的属性
- *PDO::FETCH_LAZY*：结合使用 *PDO::FETCH_BOTH* 和 *PDO::FETCH_OBJ*，创建供用来访问的对象变量名
- *PDO::FETCH_NUM*：返回一个索引为以0开始的结果集列号的数组
- *PDO::FETCH_OBJ*：返回一个属性名对应结果集列名的匿名对象

cursor_orientation 对于 一个 PDOStatement 对象表示的可滚动游标，该值决定了哪一行将被返回给调用者。此值必须是 PDO::FETCH*ORI** 系列常量中的一个，默认为 PDO::FETCH_ORI_NEXT。要想让 PDOStatement 对象使用可滚动游标，必须在用 PDO::prepare() 预处理SQL语句时，设置 PDO::ATTR_CURSOR 属性为 PDO::CURSOR_SCROLL。

offset 对于一个 cursor_orientation 参数设置为 PDO::FETCH_ORI_ABS 的PDOStatement 对象代表的可滚动游标，此值指定结果集中想要获取行的绝对行号。 对于一个 cursor_orientation 参数设置为 PDO::FETCH_ORI_REL 的PDOStatement 对象代表的可滚动游标，此值指定想要获取行相对于调用 PDOStatement::fetch() 前游标的位置

返回值

此函数（方法）成功时返回的值依赖于提取类型。在所有情况下，失败都返回 FALSE 。

实例

用不同的提取方式获取行

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* 运用 PDOStatement::fetch 风格 */
print("PDO::FETCH_ASSOC: ");
print("Return next row as an array indexed by column name\n");
$result = $sth->fetch(PDO::FETCH_ASSOC);
print_r($result);
print("\n");

print("PDO::FETCH_BOTH: ");
print("Return next row as an array indexed by both column name and number\n");
$result = $sth->fetch(PDO::FETCH_BOTH);
print_r($result);
print("\n");

print("PDO::FETCH_LAZY: ");
print("Return next row as an anonymous object with column names as properties\n");
$result = $sth->fetch(PDO::FETCH_LAZY);
print_r($result);
print("\n");

print("PDO::FETCH_OBJ: ");
print("Return next row as an anonymous object with column names as properties\n");
$result = $sth->fetch(PDO::FETCH_OBJ);
print $result->NAME;
print("\n");
?>
```

以上实例会输出：

```
PDO::FETCH_ASSOC: Return next row as an array indexed by column name
Array
(
    [NAME] => apple
    [COLOUR] => red
)

PDO::FETCH_BOTH: Return next row as an array indexed by both column name and number
Array
(
    [NAME] => banana
    [0] => banana
    [COLOUR] => yellow
    [1] => yellow
)

PDO::FETCH_LAZY: Return next row as an anonymous object with column names as properties
PDORow Object
(
    [NAME] => orange
    [COLOUR] => orange
)

PDO::FETCH_OBJ: Return next row as an anonymous object with column names as properties
kiwi
```

使用一个可滚动游标获取行

```
<?php
function readDataForwards($dbh) {
  $sql = 'SELECT hand, won, bet FROM mynumbers ORDER BY BET';
  try {
    $stmt = $dbh->prepare($sql, array(PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL));
    $stmt->execute();
    while ($row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_NEXT)) {
      $data = $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
      print $data;
    }
    $stmt = null;
  }
  catch (PDOException $e) {
    print $e->getMessage();
  }
}
function readDataBackwards($dbh) {
  $sql = 'SELECT hand, won, bet FROM mynumbers ORDER BY bet';
  try {
    $stmt = $dbh->prepare($sql, array(PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL));
    $stmt->execute();
    $row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_LAST);
    do {
      $data = $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
      print $data;
    } while ($row = $stmt->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_PRIOR));
    $stmt = null;
  }
  catch (PDOException $e) {
    print $e->getMessage();
  }
}

print "Reading forwards:\n";
readDataForwards($conn);

print "Reading backwards:\n";
readDataBackwards($conn);
?>
```

以上实例会输出：

```
Reading forwards:
21    10    5
16    0     5
19    20    10

Reading backwards:
19    20    10
16    0     5
21    10    5
```

PDOStatement::fetchAll

PDOStatement::fetchAll — 返回一个包含结果集中所有行的数组(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
array PDOStatement::fetchAll ([ int $fetch_style [, mixed $fetch_argument [, array $ctor_
```

参数

fetch_style

控制下一行如何返回给调用者。此值必须是 PDO::FETCH_* 系列常量中的一个，缺省为 PDO::ATTR_DEFAULT_FETCH_MODE 的值 （默认为 PDO::FETCH_BOTH ）。

想要返回一个包含结果集中单独一列所有值的数组，需要指定 PDO::FETCH_COLUMN 。通过指定 column-index 参数获取想要的列。

想要获取结果集中单独一列的唯一值，需要将 PDO::FETCH_COLUMN 和 PDO::FETCH_UNIQUE 按位或。

想要返回一个根据指定列把值分组后的关联数组，需要将 PDO::FETCH_COLUMN 和 PDO::FETCH_GROUP 按位或。

fetch_argument 根据 fetch_style 参数的值，此参数有不同的意义：

- `PDO::FETCH_COLUMN` ：返回指定以0开始索引的列。
- `PDO::FETCH_CLASS` ：返回指定类的实例，映射每行的列到类中对应的属性名。
- `PDO::FETCH_FUNC` ：将每行的列作为参数传递给指定的函数，并返回调用函数后的结果。

ctor_args 当 fetch_style 参数为 PDO::FETCH_CLASS 时，自定义类的构造函数的参数。

返回值

PDOStatement::fetchAll() 返回一个包含结果集中所有剩余行的数组。此数组的每一行要么是一个列值的数组，要么是属性对应每个列名的一个对象。

使用此方法获取大结果集将导致系统负担加重且可能占用大量网络资源。与其取回所有数据后用PHP来操作，倒不如考虑使用数据库服务来处理结果集。例如，在取回数据并通过PHP处理前，在 SQL 中使用 WHERE 和 ORDER BY 子句来限定结果。

实例

获取结果集中所有剩余的行

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* 获取结果集中所有剩余的行 */
print("Fetch all of the remaining rows in the result set:\n");
$result = $sth->fetchAll();
print_r($result);
?>
```

以上实例的输出为：

```
Fetch all of the remaining rows in the result set:
Array
(
    [0] => Array
        (
            [NAME] => pear
            [0] => pear
            [COLOUR] => green
            [1] => green
        )

    [1] => Array
        (
            [NAME] => watermelon
            [0] => watermelon
            [COLOUR] => pink
            [1] => pink
        )

)
```

获取结果集中单独一列的所有值

下面例子演示了如何从一个结果集中返回单独一列所有的值，尽管 SQL 语句自身可能返回每行多列。

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* 获取第一列所有值 */
$result = $sth->fetchAll(PDO::FETCH_COLUMN, 0);
var_dump($result);
?>
```

以上实例的输出为：

```
Array(3)
(
    [0] =>
    string(5) => apple
    [1] =>
    string(4) => pear
    [2] =>
    string(10) => watermelon
)
```

根据单独的一列把所有值分组

下面例子演示了如何返回一个根据结果集中指定列的值分组的关联数组。该数组包含三个键：返回的 apple 和 pear 数组包含了两种不同的颜色，而返回的 watermelon 数组仅包含一种颜色。

```
<?php
$insert = $dbh->prepare("INSERT INTO fruit(name, colour) VALUES (?, ?)");
$insert->execute(array('apple', 'green'));
$insert->execute(array('pear', 'yellow'));

$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* 根据第一列分组  */
var_dump($sth->fetchAll(PDO::FETCH_COLUMN|PDO::FETCH_GROUP));
?>
```

以上实例的输出为：

```
array(3) {
  ["apple"]=>
  array(2) {
    [0]=>
    string(5) "green"
    [1]=>
    string(3) "red"
  }
  ["pear"]=>
  array(2) {
    [0]=>
    string(5) "green"
    [1]=>
    string(6) "yellow"
  }
  ["watermelon"]=>
  array(1) {
    [0]=>
    string(5) "green"
  }
}
```

每行结果实例化一个类

下面列子演示了 PDO::FETCH_CLASS 获取风格的行为。

```
<?php
class fruit {
    public $name;
    public $colour;
}

$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

$result = $sth->fetchAll(PDO::FETCH_CLASS, "fruit");
var_dump($result);
?>
```

以上实例的输出为：

```
array(3) {
  [0]=>
  object(fruit)#1 (2) {
    ["name"]=>
    string(5) "apple"
    ["colour"]=>
    string(5) "green"
  }
  [1]=>
  object(fruit)#2 (2) {
    ["name"]=>
    string(4) "pear"
    ["colour"]=>
    string(6) "yellow"
  }
  [2]=>
  object(fruit)#3 (2) {
    ["name"]=>
    string(10) "watermelon"
    ["colour"]=>
    string(4) "pink"
  }
}
```

每行调用一次函数

下面列子演示了 PDO::FETCH_FUNC 获取风格的行为。

```
<?php
function fruit($name, $colour) {
    return "{$name}: {$colour}";
}

$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

$result = $sth->fetchAll(PDO::FETCH_FUNC, "fruit");
var_dump($result);
?>
```

以上实例的输出为：

```
array(3) {
  [0]=>
  string(12) "apple: green"
  [1]=>
  string(12) "pear: yellow"
  [2]=>
  string(16) "watermelon: pink"
}
```

PDOStatement::fetchColumn

PDOStatement::fetchColumn — 从结果集中的下一行返回单独的一列。(PHP 5 >= 5.1.0, PECL pdo >= 0.9.0)

说明

语法

```
string PDOStatement::fetchColumn ([ int $column_number = 0 ] )
```

从结果集中的下一行返回单独的一列，如果没有了，则返回 FALSE 。

参数

column_number

你想从行里取回的列的索引数字（以0开始的索引）。如果没有提供值，则 PDOStatement::fetchColumn() 获取第一列。

返回值

PDOStatement::fetchColumn() 从结果集中的下一行返回单独的一列。

注意：如果使用 PDOStatement::fetchColumn() 取回数据，则没有办法返回同一行的另外一列。**

实例

返回下一行的第一列

```
<?php
$sth = $dbh->prepare("SELECT name, colour FROM fruit");
$sth->execute();

/* 从结果集中的下一行获取第一列  */
print("从结果集中的下一行获取第一列：\n");
$result = $sth->fetchColumn();
print("name = $result\n");

print("从结果集中的下一行获取第二列：\n");
$result = $sth->fetchColumn(1);
print("colour = $result\n");
?>
```

以上实例会输出：

```
从结果集中的下一行获取第一列：
name = lemon
从结果集中的下一行获取第二列：
colour = red
```

PDOStatement::fetchObject

PDOStatement::fetchObject — 获取下一行并作为一个对象返回。(PHP 5 >= 5.1.0, PECL pdo >= 0.2.4)

说明

语法

```
mixed PDOStatement::fetchObject ([ string $class_name = "stdClass" [, array $ctor_args ]]
```

获取下一行并作为一个对象返回。此函数（方法）是使用 PDO::FETCH_CLASS 或 PDO::FETCH_OBJ 风格的 PDOStatement::fetch() 的一种替代。

参数

class_name

创建类的名称。

ctor_args

此数组的元素被传递给构造函数。

返回值

返回一个属性名对应于列名的所要求类的实例， 或者在失败时返回 FALSE。

PDOStatement::getAttribute

PDOStatement::getAttribute — 检索一个语句属性(PHP 5 >= 5.1.0, PECL pdo >= 0.2.0)

说明

语法

```
mixed PDOStatement::getAttribute ( int $attribute )
```

得到语句的一个属性。当前，不存在通用的属性，只有驱动特定的属性：

- PDO::ATTR_CURSOR_NAME （Firebird 和 ODBC 特性）： 获取 UPDATE ... WHERE CURRENT OF 的游标名称。

返回值

返回属性值。

PDOStatement::getColumnMeta

PDOStatement::getColumnMeta — 返回结果集中一列的元数据(PHP 5 >= 5.1.0, PECL pdo >= 0.2.0)

说明

语法

```
array PDOStatement::getColumnMeta ( int $column )
```

检索一个在结果集中以0开始索引的列的元数据作为一个关联数组。

注意：此函数是实验性的。此函数的表象，包括名称及其相关文档都可能在未来的 PHP 发布版本中未通知就被修改。使用本函数风险自担 。

注意：并非所有 PDO 驱动都支持 PDOStatement::getColumnMeta()。

参数

column 结果集中以0开始索引的列。

返回值

返回一个关联数组，它包含了下列表示一个单独列的元数据的值：

列的元数据 | 名称 | 值 | | --- | --- | | *native_type* | 用于表示列值的 PHP 原生类型。 | | *driver:decl_type* | 在数据库中用于表示列值的 SQL 类型。如果结果集中的列是一个函数的结果，则该值不能被 **PDOStatement::getColumnMeta()** 返回。 | | *flags* | 任何设置于此列的标记。 | | *name* | 通过数据库返回的列名。 | | *table* | 通过数据库返回的该列的表名 | | *len* | 该列的长度。除浮点小数外通常为 *-1* | | *precision* | 该列的数值精度。除浮点小数外通常为 *0* 。 | | *pdo_type* | 以 *PDO::PARAM**_ 常量为代表的列类型。 |

实例

检索列的元数据

下面例子展示了在一个PDO_SQLITE中，检索一个通过函数（COUNT）生成单独列的元数据的结果。

```
<?php
$select = $DB->query('SELECT COUNT(*) FROM fruit');
$meta = $select->getColumnMeta(0);
var_dump($meta);
?>
```

以上实例输出：

```
array(6) {
  ["native_type"]=>
  string(7) "integer"
  ["flags"]=>
  array(0) {
  }
  ["name"]=>
  string(8) "COUNT(*)"
  ["len"]=>
  int(-1)
  ["precision"]=>
  int(0)
  ["pdo_type"]=>
  int(2)
}
```

PDOStatement::nextRowset

PDOStatement::nextRowset — 在一个多行集语句句柄中推进到下一个行集(PHP 5 >= 5.1.0, PECL pdo >= 0.2.0)

说明

语法

```
bool PDOStatement::nextRowset ( void )
```

一些数据库服务支持返回一个以上行集（也被称为结果集）的存储过程。

PDOStatement::nextRowset() 使你能够结合一个 PDOStatement 对象访问第二个以及后续的行集。上述的每个行集可以有不同的列集合。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

获取由一个存储过程返回的多个行集

下面例子展示了怎样调用一个存储过程，返回三个行集的 MULTIPLE_ROWSETS 。用一个 do / while 循环来循环调用 PDOStatement::nextRowset() 方法， 当不再有行集返回时返回 false 并结束循环。

```
<?php
$sql = 'CALL multiple_rowsets()';
$stmt = $conn->query($sql);
$i = 1;
do {
    $rowset = $stmt->fetchAll(PDO::FETCH_NUM);
    if ($rowset) {
        printResultSet($rowset, $i);
    }
    $i++;
} while ($stmt->nextRowset());

function printResultSet(&$rowset, $i) {
    print "Result set $i:\n";
    foreach ($rowset as $row) {
        foreach ($row as $col) {
            print $col . "\t";
        }
        print "\n";
    }
    print "\n";
}
?>
```

以上实例输出：

```
Result set 1:
apple    red
banana   yellow

Result set 2:
orange   orange    150
banana   yellow    175

Result set 3:
lime     green
apple    red
banana   yellow
```

PDOStatement::rowCount

PDOStatement::rowCount — 返回受上一个 SQL 语句影响的行数(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

说明

语法

```
int PDOStatement::rowCount ( void )
```

PDOStatement::rowCount() 返回上一个由对应的 PDOStatement 对象执行DELETE、INSERT、或 UPDATE 语句受影响的行数。

如果上一条由相关 PDOStatement 执行的 SQL 语句是一条 SELECT 语句，有些数据可能返回由此语句返回的行数。但这种方式不能保证对所有数据有效，且对于可移植的应用不应依赖于此方式。

返回值

返回行数。

实例

返回删除的行数

PDOStatement::rowCount() 返回受 DELETE、INSERT、 或 UPDATE 语句影响的行数。

```
<?php
/*  从 FRUIT 数据表中删除所有行 */
$del = $dbh->prepare('DELETE FROM fruit');
$del->execute();

/*  返回被删除的行数 */
print("Return number of rows that were deleted:\n");
$count = $del->rowCount();
print("Deleted $count rows.\n");
?>
```

以上实例输出：

```
Return number of rows that were deleted:
Deleted 9 rows.
```

计算由一个 SELECT 语句返回的行数

对于大多数数据库，PDOStatement::rowCount() 不能返回受一条 SELECT 语句影响的行数。替代的方法是，使用 PDO::query() 来发出一条和原打算中的SELECT语句有相同条件表达式的 SELECT COUNT(*) 语句，然后用 PDOStatement::fetchColumn() 来取得返回的行数。这样应用程序才能正确执行。

```
<?php
$sql = "SELECT COUNT(*) FROM fruit WHERE calories > 100";
if ($res = $conn->query($sql)) {

    /* 检查符合 SELECT 语句的行数 */
  if ($res->fetchColumn() > 0) {

        /* 发出一条真正的 SELECT 语句并操作返回的结果 */
         $sql = "SELECT name FROM fruit WHERE calories > 100";
       foreach ($conn->query($sql) as $row) {
           print "Name: " .  $row['NAME'] . "\n";
         }
    }
    /* 没有匹配的行 -- 执行其他 */
  else {
      print "No rows matched the query.";
    }
}

$res = null;
$conn = null;
?>
```

以上实例输出结果为：

```
apple
banana
orange
pear
```

PDOStatement::setAttribute

PDOStatement::setAttribute — 设置一个语句属性(PHP 5 >= 5.1.0, PECL pdo >= 0.2.0)

说明

语法

```
bool PDOStatement::setAttribute ( int $attribute , mixed $value )
```

给语句设置一个属性。当前，没有通用的属性可以设置，只有驱动特定的属性：

- *PDO::ATTR_CURSOR_NAME* （Firebird 和 ODBC 特性）： 为 *UPDATE ... WHERE CURRENT OF* 设置游标名称。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

PDOStatement::setFetchMode

PDOStatement::setFetchMode — 为语句设置默认的获取模式。(PHP 5 >= 5.1.0, PECL pdo >= 0.2.0)

说明

语法

```
bool PDOStatement::setFetchMode ( int $mode )
```

```
bool PDOStatement::setFetchMode ( int $PDO::FETCH_COLUMN , int $colno )
```

```
bool PDOStatement::setFetchMode ( int $PDO::FETCH_CLASS , string $classname , array $ctor
```

```
bool PDOStatement::setFetchMode ( int $PDO::FETCH_INTO , object $object )
```

参数

mode 获取模式必须是 PDO::FETCH_* 系列常量中的一个。

colno 列号。

classname 类名。

ctorargs 构造函数参数。

object 对象。

返回值

成功时返回 TRUE， 或者在失败时返回 FALSE。

实例

设置获取模式

下面的例子示范如何用 PDOStatement::setFetchMode() 来为一个 PDOStatement 对象更改默认的获取模式。

```
<?php
$sql = 'SELECT name, colour, calories FROM fruit';
try {
  $stmt = $dbh->query($sql);
  $result = $stmt->setFetchMode(PDO::FETCH_NUM);
  while ($row = $stmt->fetch()) {
    print $row[0] . "\t" . $row[1] . "\t" . $row[2] . "\n";
  }
}
catch (PDOException $e) {
  print $e->getMessage();
}
?>
```

以上实例输出为：

```
apple   red     150
banana  yellow  250
orange  orange  300
kiwi    brown   75
lemon   yellow  25
pear    green   150
watermelon      pink    90
```

PHP SimpleXML 函数

PHP SimpleXML 简介

SimpleXML 函数允许您把 XML 转换为对象。

通过普通的属性选择器或数组迭代器，可以处理这个对象，就像处理任何其他对象一样。

其中的一些函数需要最新的 PHP 版本。

安装

SimpleXML 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP SimpleXML 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
__construct()	创建一个新的 SimpleXMLElement 对象。	5
addAttribute()	给 SimpleXML 元素添加一个属性。	5
addChild()	给 SimpleXML 元素添加一个子元素。	5
asXML()	从 SimpleXML 元素获取 XML 字符串。	5
attributes()	获取 SimpleXML 元素的属性。	5
children()	获取指定节点的子。	5
getDocNamespaces()	获取 XML 文档的命名空间。	5
getName()	获取 SimpleXML 元素的名称。	5
getNamespaces()	从 XML 数据获取命名空间。	5
registerXPathNamespace()	为下一次 XPath 查询创建命名空间语境。	5
simplexml_import_dom()	从 DOM 节点获取 SimpleXMLElement 对象。	5
simplexml_load_file()	从 XML 文档获取 SimpleXMLElement 对象。	5
simplexml_load_string()	从 XML 字符串获取 SimpleXMLElement 对象。	5
xpath()	对 XML 数据运行 XPath 查询。	5

PHP SimpleXML 常量

无。

PHP __construct() 函数

定义和用法

__construct() 函数创建一个新的 SimpleXMLElement 对象。

如果成功，则该函数返回一个对象。如果失败，则返回 false。

语法

```
__construct(data,options,is_url,ns,is_prefix)
```

参数	描述
data	必需。形式良好的 XML 字符串或 XML 文档的路径或 URL。
options	可选。规定附加的 Libxml 参数。
is_url	可选。规定 data 参数是否是 URL。默认是 false。
ns	可选。
is_prefix	可选。

返回值

返回一个表示数据的 SimpleXMLElement 对象。

例子

```
<?php
$xmlstring = <<<XML
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
XML;

$xml = new SimpleXMLElement($xmlstring);

echo $xml->body[0];
?>
```

输出类似：

```
Don't forget the meeting!
```

PHP addAttribute() 函数

定义和用法

addAttribute() 函数给 SimpleXML 元素添加一个属性。

该函数无返回值。

语法

```
class SimpleXMLElement
{
string addAttribute(name,value,ns)
}
```

参数	描述
name	必需。规定属性的名称。
value	必需。规定属性的值。
ns	可选。规定属性的命名空间。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
$xml = simplexml_load_file("test.xml");

$xml->body[0]->addAttribute("type", "small");

foreach($xml->body[0]->attributes() as $a => $b)
  {
  echo $a,'="',$b,'"';
  }
?>
```

输出：

```
type="small"
```

PHP addChild() 函数

定义和用法

addChild() 函数向指定的 XML 节点添加一个子节点。

该函数返回一个 SimpleXMLElement 对象，这个对象表示添加到 XML 节点的子元素。

语法

```
class SimpleXMLElement
{
string addChild(name,value,ns)
}
```

参数	描述
name	必需。规定子元素的名称。
value	必需。规定子元素的值。
ns	可选。规定子元素的命名空间。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
$xml = simplexml_load_file("test.xml");

$xml->body[0]->addChild("date", "2008-08-08");

foreach ($xml->body->children() as $child)
  {
  echo "Child node: " . $child;
  }
?>
```

输出：

```
Child node: 2008-08-08
```

PHP asXML() 函数

定义和用法

asXML() 函数以字符串的形式从 SimpleXMLElement 对象返回 XML 文档。

若失败，则返回 false。

语法

```
class SimpleXMLElement
{
string asXML(file)
}
```

参数	描述
file	可选。如果规定了此参数，则该函数会把 XML 写入一个文件，而不是返回它。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
if (file_exists('test.xml'))
  {
  $xml = simplexml_load_file('test.xml');
  }

echo $xml->asXML();
?>
```

输出：

```
George John Reminder Don't forget the meeting!
```

如果在浏览器窗口中选择“查看源文件”，会看到这些 HTML：

```
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP attributes() 函数

定义和用法

attributes() 函数获取 SimpleXML 元素的属性。

该函数提供在一个 XML 标签中定义的属性和值。

语法

```
class SimpleXMLElement
{
string attributes(_ns_,_is_prefix_)
}
```

参数	描述
ns	可选。被检索的属性的命名空间。
is_prefix	可选。默认是 false。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body type="small" important="low">Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
$xml = simplexml_load_file("test.xml");

foreach($xml->body[0]->attributes() as $a => $b)
  {
  echo $a,'="',$b,'"';
  }
?>
```

输出：

```
type="small" important="low"
```

PHP children() 函数

定义和用法

children() 函数获取指定节点的子节点。

语法

```
class SimpleXMLElement
{
string children(ns,is_prefix)
}
```

参数	描述
ns	可选。
is_prefix	可选。默认是 false。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
$xml = simplexml_load_file("test.xml");

foreach ($xml->children() as $child)
  {
  echo "Child node: " . $child;
  }
?>
```

输出类似：

```
Child node: George
Child node: John
Child node: Reminder
Child node: Don't forget the meeting!
```

PHP getDocNamespaces() 函数

定义和用法

getDocNamespaces() 函数从 SimpleXMLElement 对象返回在 XML 文档中声明的命名空间。

如果成功，该函数返回包含命名空间名称（带有关联的 URL）的数组。如果失败，则返回 false。

语法

```
class SimpleXMLElement
{
string getDocNamespaces(recursive)
}
```

参数	描述
recursive	可选。规定是否返回父子节点中的所有命名空间。默认是 false。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note xmlns:b="http://www.w3school.com.cn/example/">
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<b:body>Don't forget the meeting!</b:body>
</note>
```

PHP 代码：

```
<?php
if (file_exists('test.xml'))
  {
  $xml = simplexml_load_file('test.xml');
  }

print_r($xml->getDocNamespaces());
?>
```

输出类似：

```
Array
(
[b] => http://www.w3school.com.cn/example/
)
```

PHP getName() 函数

定义和用法

getName() 函数从 SimpleXMLElement 对象获取 XML 元素的名称。

如果成功，该函数返回当前的 XML 元素的名称。如果失败，则返回 false。

语法

```
class SimpleXMLElement
{
string getName()
}
```

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<b:body>Don't forget the meeting!</b:body>
</note>
```

PHP 代码：

```
<?php
if (file_exists('test.xml'))
  {
  $xml = simplexml_load_file('test.xml');
  }

echo $xml->getName();

foreach($xml->children() as $child)
  {
  echo $child->getName();
  }
?>
```

输出类似：

```
note
to
from
heading
body
```

PHP getNamespace() 函数

定义和用法

getNamespace() 函数获取在 XML 文档中使用的命名空间。

如果成功，该函数返回命名空间（带有关联的 URL）的一个数组。如果失败，则返回 false。

语法

```
class SimpleXMLElement
{
string getNamespace(recursive)
}
```

参数	描述
recursive	可选。规定是否返回父子节点中使用的所有命名空间。默认是 false。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note xmlns:b="http://www.w3school.com.cn/example/">
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<b:body>Don't forget the meeting!</b:body>
</note>
```

PHP 代码：

```
<?php
if (file_exists('test.xml'))
  {
  $xml = simplexml_load_file('test.xml');
  }

print_r($xml->getNamespaces());
?>
```

输出类似：

```
Array
(
[b] => http://www.w3school.com.cn/example/
)
```

PHP registerXPathNamespace() 函数

定义和用法

registerXPathNamespace() 函数为下一次 XPath 查询创建命名空间语境。

语法

```
class SimpleXMLElement
{
string registerXPathNamespace(prefix,ns)
}
```

参数	描述
prefix	必需。规定命名空间前缀。
ns	必需。规定命名空间 URL。必须匹配 XML 文档中的命名空间。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note xmlns:b="http://www.w3school.com.cn/example/">
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<b:body>Don't forget the meeting!</b:body>
</note>
```

PHP 代码：

```
<?php
$xml = simplexml_load_file("test.xml");

$xml->registerXPathNamespace("msg", "http://www.w3school.com.cn/example/");
$result = $xml->xpath("msg:body");

foreach ($result as $message)
  {
  echo $message;
  }
?>
```

输出类似：

```
Don't forget the meeting!
```

PHP simplexml_import_dom() 函数

定义和用法

simplexml_import_dom() 函数把 DOM 节点转换为 SimpleXMLElement 对象。

如果失败，则该函数返回 false。

语法

```
simplexml_import_dom(data,class)
```

参数	描述
data	必需。规定要使用的 DOM 节点。
class	必需。规定新对象的 class。

例子

```
<?php
$dom = new domDocument;
$dom->loadXML('<note><from>John</from></note>');

$xml = simplexml_import_dom($dom);

echo $xml->from;
?>
```

输出类似：

```
John
```

PHP simplexml_load_file() 函数

定义和用法

simplexml_load_file() 函数把 XML 文档载入对象中。

如果失败，则返回 false。

语法

```
simplexml_load_file(file,class,options,ns,is_prefix)
```

参数	描述
file	必需。规定要使用的 XML 文档。
class	可选。规定新对象的 class。
options	可选。规定附加的 Libxml 参数。
ns	可选。
is_prefix	可选。

返回值

返回类 SimpleXMLElement 的一个对象，该对象的属性包含 XML 文档中的数据。如果失败，则返回 false。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
if (file_exists('test.xml'))
  {
  $xml = simplexml_load_file('test.xml');
  var_dump($xml);
  }

else
  {
  exit('Error.');
  }
?>
```

输出：

```
object(SimpleXMLElement)#1 (4)
{
["to"]=> string(4) "George"
["from"]=> string(4) "John"
["heading"]=> string(8) "Reminder"
["body"]=> string(29) "Don't forget the meeting!"
}
```

PHP simplexml_load_string() 函数

定义和用法

simplexml_load_string() 函数把 XML 字符串载入对象中。

如果失败，则返回 false。

语法

```
simplexml_load_file(string,class,options,ns,is_prefix)
```

参数	描述
string	必需。规定要使用的 XML 字符串。
class	可选。规定新对象的 class。
options	可选。规定附加的 Libxml 参数。
ns	可选。
is_prefix	可选。

返回值

返回类 SimpleXMLElement 的一个对象，该对象的属性包含 XML 文档中的数据。如果失败，则返回 false。

例子

```
<?php
$xmlstring = <<<XML
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
XML;

$xml = simplexml_load_string($xmlstring);

var_dump($xml);
?>
```

输出：

```
object(SimpleXMLElement)#1 (4)
{
["to"]=> string(4) "George"
["from"]=> string(4) "John"
["heading"]=> string(8) "Reminder"
["body"]=> string(29) "Don't forget the meeting!"
}
```

PHP xpath() 函数

定义和用法

xpath() 函数运行对 XML 文档的 XPath 查询。

如果成功，则返回包含 SimpleXMLElement 对象的一个数组。如果失败，则返回 false。

语法

```
class SimpleXMLElement
{
string xpath(path)
}
```

参数	描述
path	必需。XPath 路径。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
$xml = simplexml_load_file("test.xml");

$result = $xml->xpath("from");

print_r($result);
?>
```

输出：

```
Array
(
[0] => SimpleXMLElement Object
  (
  [0] => John
  )
)
```

PHP String 函数

PHP String 简介

String 字符串函数允许您对字符串进行操作。

安装

String 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP String 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
addcslashes()	在指定的字符前添加反斜杠。	4
addslashes()	在指定的预定义字符前添加反斜杠。	3
bin2hex()	把 ASCII 字符的字符串转换为十六进制值。	3
chop()	rtrim() 的别名。	3
chr()	从指定的 ASCII 值返回字符。	3
chunk_split()	把字符串分割为一连串更小的部分。	3
convert_cyr_string()	把字符由一种 Cyrillic 字符转换成另一种。	3
convert_uudecode()	对 uuencode 编码的字符串进行解码。	5
convert_uuencode()	使用 uuencode 算法对字符串进行编码。	5
count_chars()	返回字符串所用字符的信息。	4
crc32()	计算一个字符串的 32-bit CRC。	4
crypt()	单向的字符串加密法 (hashing)。	3
echo()	输出字符串。	3
explode()	把字符串打散为数组。	3
fprintf()	把格式化的字符串写到指定的输出流。	5
get_html_translation_table()	返回翻译表。	4
hebrev()	把希伯来文本从右至左的流转换为左至右的流。	3

hebrevc()	同上，同时把(\n) 转为 。	3
html_entity_decode()	把 HTML 实体转换为字符。	4
htmlentities()	把字符转换为 HTML 实体。	3
htmlspecialchars_decode()	把一些预定义的 HTML 实体转换为字符。	5
htmlspecialchars()	把一些预定义的字符转换为 HTML 实体。	3
implode()	把数组元素组合为一个字符串。	3
join()	implode() 的别名。	3
levenshtein()	返回两个字符串之间的 Levenshtein 距离。	3
localeconv()	返回包含本地数字及货币信息格式的数组。	4
ltrim()	从字符串左侧删除空格或其他预定义字符。	3
md5()	计算字符串的 MD5 散列。	3
md5_file()	计算文件的 MD5 散列。	4
metaphone()	计算字符串的 metaphone 键。	4
money_format()	把字符串格式化为货币字符串。	4
nl_langinfo()	返回指定的本地信息。	4
nl2br()	在字符串中的每个新行之前插入 HTML 换行符。	3
number_format()	通过千位分组来格式化数字。	3
ord()	返回字符串第一个字符的 ASCII 值。	3
parse_str()	把查询字符串解析到变量中。	3
print()	输出一个或多个字符串。	3
printf()	输出格式化的字符串。	3
quoted_printable_decode()	解码 quoted-printable 字符串。	3
quotemeta()	在字符串中某些预定义的字符前添加反斜杠。	3
rtrim()	从字符串的末端开始删除空白字符或其他预定义字符。	3
setlocale()	设置地区信息（地域信息）。	3
sha1()	计算字符串的 SHA-1 散列。	4
sha1_file()	计算文件的 SHA-1 散列。	4
similar_text()	计算两个字符串的匹配字符的数目。	3
soundex()	计算字符串的 soundex 键。	3
sprintf()	把格式化的字符串写写入一个变量中。	3
sscanf()	根据指定的格式解析来自一个字符串的输入。	4

str_ireplace()	替换字符串中的一些字符。（对大小写不敏感）	5
str_pad()	把字符串填充为新的长度。	4
str_repeat()	把字符串重复指定的次数。	4
str_replace()	替换字符串中的一些字符。（对大小写敏感）	3
str_rot13()	对字符串执行 ROT13 编码。	4
str_shuffle()	随机地打乱字符串中的所有字符。	4
str_split()	把字符串分割到数组中。	5
str_word_count()	计算字符串中的单词数。	4
strcasecmp()	比较两个字符串。（对大小写不敏感）	3
strchr()	搜索字符串在另一字符串中的第一次出现。strstr() 的别名	3
strcmp()	比较两个字符串。（对大小写敏感）	3
strcoll()	比较两个字符串（根据本地设置）。	4
strcspn()	返回在找到任何指定的字符之前，在字符串查找的字符数。	3
strip_tags()	剥去 HTML、XML 以及 PHP 的标签。	3
stripcslashes()	删除由 addcslashes() 函数添加的反斜杠。	4
stripslashes()	删除由 addslashes() 函数添加的反斜杠。	3
stripos()	返回字符串在另一字符串中第一次出现的位置(大小写不敏感)	5
stristr()	查找字符串在另一字符串中第一次出现的位置(大小写不敏感)	3
strlen()	返回字符串的长度。	3
strnatcasecmp()	使用一种“自然”算法来比较两个字符串（对大小写不敏感）	4
strnatcmp()	使用一种“自然”算法来比较两个字符串（对大小写敏感）	4
strncasecmp()	前 n 个字符的字符串比较（对大小写不敏感）。	4
strncmp()	前 n 个字符的字符串比较（对大小写敏感）。	4
strpbrk()	在字符串中搜索指定字符中的任意一个。	5
strpos()	返回字符串在另一字符串中首次出现的位置（对大小写敏感）	3
strrchr()	查找字符串在另一个字符串中最后一次出现的位置。	3
strrev()	反转字符串。	3

strripos()	查找字符串在另一字符串中最后出现的位置(对大小写不敏感)	5
strrpos()	查找字符串在另一字符串中最后出现的位置(对大小写敏感)	3
strspn()	返回在字符串中包含的特定字符的数目。	3
strstr()	搜索字符串在另一字符串中的首次出现（对大小写敏感）	3
strtok()	把字符串分割为更小的字符串。	3
strtolower()	把字符串转换为小写。	3
strtoupper()	把字符串转换为大写。	3
strtr()	转换字符串中特定的字符。	3
substr()	返回字符串的一部分。	3
substr_compare()	从指定的开始长度比较两个字符串。	5
substr_count()	计算子串在字符串中出现的次数。	4
substr_replace()	把字符串的一部分替换为另一个字符串。	4
trim()	从字符串的两端删除空白字符和其他预定义字符。	3
ucfirst()	把字符串中的首字符转换为大写。	3
ucwords()	把字符串中每个单词的首字符转换为大写。	3
vfprintf()	把格式化的字符串写到指定的输出流。	5
vprintf()	输出格式化的字符串。	4
vsprintf()	把格式化字符串写入变量中。	4
wordwrap()	按照指定长度对字符串进行折行处理。	4

PHP String 常量

PHP：指示支持该常量的最早的 PHP 版本。

常量	描述	PHP
CRYPT_SALT_LENGTH	包含系统默认加密方法的长度。对于标准 DES 加密，长度是 2。	
CRYPT_STD_DES	如果支持 2 字符 salt 的 DES 加密，则设置为 1，否则为 0。	
CRYPT_EXT_DES	如果支持 9 字符 salt 的 DES 加密，则设置为 1，否则为 0。	
CRYPT_MD5	如果支持以1开始的 12 字符 salt 的MD5加密，则设置为1，否则为0。	
CRYPT_BLOWFISH	如果支持以 2 或 $2a$ 开始的 16 字符 salt 的 Blowfish 加密，则设置为 1，否则为 0。	
HTML_SPECIALCHARS		
HTML_ENTITIES		
ENT_COMPAT		
ENT_QUOTES		
ENT_NOQUOTES		
CHAR_MAX		
LC_CTYPE		
LC_NUMERIC		
LC_TIME		
LC_COLLATE		
LC_MONETARY		
LC_ALL		
LC_MESSAGES		
STR_PAD_LEFT		
STR_PAD_RIGHT		
STR_PAD_BOTH		

PHP addcslashes() 函数

定义和用法

addcslashes() 函数在指定的字符前添加反斜杠。

语法

```
addcslashes(_string_,_characters_)
```

参数	描述
string	必需。规定要检查的字符串。
characters	可选。规定受 addcslashes() 影响的字符或字符范围。

提示和注释

注释：在对 0，r，n 和 t 应用 addcslashes() 时要小心。在 PHP 中，\0，\r，\n 和 \t 是预定义的转义序列。

实例

例子 1

在本例中，我们要向字符串中的特定字符添加反斜杠：

```
<?php
$str = "Hello, my name is John Adams.";
echo $str;
echo addcslashes($str,'m');
echo addcslashes($str,'J');
?>
```

输出：

```
Hello, my name is John Adams.
Hello, \my na\me is John Ada\ms.
Hello, my name is \John Adams.
```

例子 2

在本例中，我们要向字符串中的一个范围内的字符添加反斜杠：

```
<?php
$str = "Hello, my name is John Adams.";
echo $str;
echo addcslashes($str,'A..Z');
echo addcslashes($str,'a..z');
echo addcslashes($str,'a..h');
?>
```

输出：

```
Hello, my name is John Adams.
\Hello, my name is \John \Adams.
H\e\l\l\o, \m\y \n\a\m\e \i\s J\o\h\n A\d\a\m\s.
H\ello, my n\am\e is Jo\hn A\d\ams.
```

PHP addslashes() 函数

定义和用法

addslashes() 函数在指定的预定义字符前添加反斜杠。

这些预定义字符是：

- 单引号 (')
- 双引号 (")
- 反斜杠 ()
- NULL

语法

```
addslashes(string)
```

参数	描述
string	必需。规定要检查的字符串。

提示和注释

提示：该函数可用于为存储在数据库中的字符串以及数据库查询语句准备合适的字符串。

注释：默认情况下，PHP 指令 magic_quotes_gpc 为 on，对所有的 GET、POST 和 COOKIE 数据自动运行 addslashes()。不要对已经被 magic_quotes_gpc 转义过的字符串使用 addslashes()，因为这样会导致双层转义。遇到这种情况时可以使用函数 get_magic_quotes_gpc() 进行检测。

例子

在本例中，我们要向字符串中的预定义字符添加反斜杠：

```
<?php
$str = "Who's John Adams?";
echo $str . " This is not safe in a database query.<br />";
echo addslashes($str) . " This is safe in a database query.";
?>
```

输出：

```
Who's John Adams? This is not safe in a database query.
Who\'s John Adams? This is safe in a database query.
```

PHP bin2hex() 函数

定义和用法

bin2hex() 函数把 ASCII 字符的字符串转换为十六进制值。

语法

```
bin2hex(string)
```

参数	描述
string	必需。规定要转换的字符串。

例子

在本例中，我们将把一个字符串值从二进制转换为十六进制，再转换回去：

```
<?php
$str = "Hello world!";
echo bin2hex($str);
echo pack("H*",bin2hex($str));
?>
```

输出：

```
48656c6c6f20776f726c6421
Hello world!
```

PHP chop() 函数

定义和用法

chop() 函数从字符串的末端开始删除空白字符或其他预定义字符。

该函数的 rtrim() 函数的别名。

语法

```
chop(string,charlist)
```

参数	描述
string	必需。规定要转换的字符串。
charlist	可选。规定从字符串中删除哪些字符。如果未设置该参数，则全部删除以下字符： " `\0` " - ASCII 0, NULL " `\t` " - ASCII 9, 制表符 " `\n` " - ASCII 10, 新行 " `\x0B` " - ASCII 11, 垂直制表符 " `\r` " - ASCII 13, 回车 " " - ASCII 32, 空格

例子

在本例中，我们将使用 chop() 函数从字符串右端删除字符：

```
<?php
$str = "Hello World!\n\n";
echo $str;
echo chop($str);
?>
```

以上代码输出的源代码：

```
<html>

<body>
Hello World!

Hello World!</body>

</html>
```

输出：

```
Hello World! Hello World!
```

PHP chr() 函数

定义和用法

chr() 函数从指定的 ASCII 值返回字符。

语法

```
chr(ascii)
```

参数	描述
ascii	必需。ASCII 值。

提示和注释

注释：ascii 参数可以是十进制、八进制或十六进制。通过前置 0 来规定八进制，通过前置 0x 来规定十六进制。

例子

```
<?php
echo chr(52);
echo chr(052);
echo chr(0x52);
?>
```

输出：

```
4
*
R
```

PHP chunk_split() 函数

定义和用法

chunk_split() 函数把字符串分割为一连串更小的部分。

语法

```
chunk_split(string,length,end)
```

参数	描述
string	必需。规定要分割的字符串。
length	可选。一个数字，定义字符串块的长度。
end	可选。字符串值，定义在每个字符串块之后放置的内容。

提示和注释

注释：本函数不改变原始字符串。

例子

例子 1

本例分隔每个字符，并添加 "."：

```
<?php
$str = "Hello world!";
echo chunk_split($str,1,".");
?>
```

输出：

```
H.e.l.l.o. .w.o.r.l.d.!.
```

例子 2

本例将在六个字符之后分割一次字符串，并添加 "..."：

```
<?php
$str = "Hello world!";
echo chunk_split($str,6,"...");
?>
```

输出：

```
Hello ...world!...
```

PHP convert_cyr_string() 函数

定义和用法

convert_cyr_string() 函数把字符由一种 Cyrillic 字符转换成另一种。

被支持的 Cyrillic 字符集是：

- k - koi8-r
- w - windows-1251
- i - iso8859-5
- a - x-cp866
- d - x-cp866
- m - x-mac-cyrillic

语法

```
convert_cyr_string(string,from,to)
```

参数	描述
string	必需。规定要转换的字符串。
from	必需。源 Cyrillic 字符集。
to	必需。目标 Cyrillic 字符集。

提示和注释

注释：本函数可安全用于二进制对象。

PHP convert_uudecode() 函数

定义和用法

convert_uudecode() 函数对 uuencode 编码的字符串进行解码。

语法

```
convert_uudecode(string)
```

参数	描述
string	必需。规定要解码的字符串。

例子

在本例中，我们将通过使用 convert_uudecode() 对 uuencode 编码的字符串进行解码：

```
<?php
$str = ",2&5L;&\@=V]R;&0A `";
echo convert_uudecode($str);
?>
```

输出：

```
Hello world!
```

PHP convert_uuencode() 函数

定义和用法

convert_uuencode() 函数使用 uuencode 算法对字符串进行编码。

语法

```
convert_uuencode(string)
```

参数	描述
string	必需。规定进行 uuencode 的字符串。

提示和注释

注释：本函数把所有字符串（包括二进制的）转换为可打印的字符串，确保其网络传输的安全。

注释：uuencode 的字符串比原字符串增大大约 35%。

例子

在本例中，我们将使用 convert_uuencode() 对字符串进行编码：

```
<?php
$str = "Hello world!";
echo convert_uuencode($str);
?>
```

输出：

```
,2&5L;&\@=V]R;&0A `
```

PHP count_chars() 函数

定义和用法

count_chars() 函数返回字符串所用字符的信息。

语法

```
count_chars(_string_,_mode_)
```

参数	描述
string	必需。规定要检查的字符串。
mode	可选。规定返回模式。默认是 0。有以下不同的返回模式： `0` - 数组，ASCII 值为键名，出现的次数为键值 `1` - 数组，ASCII 值为键名，出现的次数为键值，只列出出现次数大于 0 的值 `2` - 数组，ASCII 值为键名，出现的次数为键值，只列出出现次数等于 0 的值 `3` - 字符串，带有所有使用过的不同的字符 `4` - 字符串，带有所有未使用过的不同的字符

实例

例子 1

在本例中，我们将使用 count_chars() 来检查字符串，返回模式设置为 1：

```
<?php
$str = "Hello World!";
print_r(count_chars($str,1));
?>
```

输出：

```
Array
(
[32] => 1
[33] => 1
[72] => 1
[87] => 1
[100] => 1
[101] => 1
[108] => 3
[111] => 2
[114] => 1
)
```

例子 2

在本例中，我们将使用 count_chars() 来检查字符串，返回模式设置为 3：

```
<?php
$str = "Hello World!";
echo count_chars($str,3);
?>
```

输出：

```
 !HWdelor
```

PHP crc32() 函数

定义和用法

crc32() 函数计算一个字符串的 crc32 多项式。

该函数可用于验证数据的完整性。

语法

```
crc32(string)
```

参数	描述
string	必需。规定要计算的字符串。

说明

生成 string 参数的 32 位循环冗余校验码多项式。这通常用于检查传输的数据是否完整。

提示和注释

提示：由于 PHP 的整数是带符号的，许多 crc32 校验码将返回负整数，因此您需要使用 sprintf() 或 printf() 的 "%u" 格式符来获取表示无符号 crc32 校验码的字符串。

例子

例子 1

在本例中，我们将在使用以及不使用 "%u" 格式符的情况下，输出 crc32() 的结果（注意结果是相同的）：

```
<?php
$str = crc32("Hello world!");
echo 'Without %u: '.$str."<br />";
echo 'With %u: ';
printf("%u",$str);
?>
```

输出：

```
Without %u: 461707669
With %u: 461707669
```

例子 2

在本例中，我们将在使用以及不使用 "%u" 格式符的情况下，输出 crc32() 的结果（注意结果是不相同的）：

```
<?php
$str = crc32("Hello world.");
echo 'Without %u: '.$str."<br />";
echo 'With %u: ';
printf("%u",$str);
?>
```

输出：

```
Without %u: -1959132156
With %u: 2335835140
```

PHP crypt() 函数

定义和用法

crypt() 函数返回使用 DES、Blowfish 或 MD5 加密的字符串。

在不同的操作系统上，本函数的行为不同，某些操作系统支持一种以上的算法类型。在安装时，PHP 会检查什么算法可用以及使用什么算法。

语法

```
crypt(str,salt)
```

参数	描述
str	必需。规定要编码的字符串。
salt	可选。用于增加被编码字符数目的字符串，以使编码更加安全。如果未提供 salt 参数，则每次调用该函数时会随机生成一个。

说明

确切的算法依赖于 salt 参数的格式和长度。

下面是与 crypt() 函数一起使用的一些常量。在安装时，由 PHP 设置这些常量：

- [CRYPT_SALT_LENGTH]
- [CRYPT_STD_DES]
- [CRYPT_EXT_DES]
- [CRYPT_MD5]
- [CRYPT_BLOWFISH]

提示和注释

提示：解密函数是没有的。crypt() 函数使用一种单向算法。

例子

在本例中，我们将测试不同的算法：

```
<?php
if (CRYPT_STD_DES == 1)
{
echo "Standard DES: ".crypt("hello world")."\n<br />";
}
else
{
echo "Standard DES not supported.\n<br />";
}

if (CRYPT_EXT_DES == 1)
{
echo "Extended DES: ".crypt("hello world")."\n<br />";
}
else
{
echo "Extended DES not supported.\n<br />";
}

if (CRYPT_MD5 == 1)
{
echo "MD5: ".crypt("hello world")."\n<br />";
}
else
{
echo "MD5 not supported.\n<br />";
}

if (CRYPT_BLOWFISH == 1)
{
echo "Blowfish: ".crypt("hello world");
}
else
{
echo "Blowfish DES not supported.";
}
?>
```

输出类似（依赖于操作系统）：

```
Standard DES: $1$r35.Y52.$iyiFuvM.zFGsscpU0aZ4e.
Extended DES not supported.
MD5: $1$BN1.0I2.$8oBI/4mufxK6Tq89M12mk/
Blowfish DES not supported.
```

PHP echo() 函数

定义和用法

echo() 函数输出一个或多个字符串。

语法

```
echo(strings)
```

参数	描述
strings	必需。一个或多个要发送到输出的字符串。

提示和注释

注释：echo() 实际上不是一个函数，因此您无需对其使用括号。不过，如果您希望向 echo() 传递一个或多个参数，那么使用括号会发生解析错误。

提示：echo() 函数比 print() 函数快一点点。

提示：echo() 函数可以使用简化语法。参见例子 5。

例子

例子 1

```
<?php
$str = "Who's John Adams?";
echo $str;
echo "<br />";
echo $str."<br />I don't know!";
?>
```

输出：

```
Who's John Adam?
Who's John Adam?
I don't know!
```

例子 2

```
<?php
echo "This text
spans multiple
lines.";
?>
```

输出：

```
This text spans multiple lines.
```

例子 3

```
<?php
echo 'This ','string ','was ','made ','with multiple parameters';
?>
```

输出：

```
This string was made with multiple parameters
```

例子 4

单引号和双引号的不同之处。单引号仅输出变量名，而不是值：

```
<?php
$color = "red";
echo "Roses are $color";
echo "<br />";
echo 'Roses are $color';
?>
```

输出：

```
Roses are red
Roses are $color
```

例子 5

简化语法：

```
<html>
<body>

<?php
$color = "red";
?>

<p>Roses are <?=$color?></p>

</body>
</html>
```

PHP explode() 函数

定义和用法

explode() 函数把字符串分割为数组。

语法

```
explode(separator,string,limit)
```

参数	描述
separator	必需。规定在哪里分割字符串。
string	必需。要分割的字符串。
limit	可选。规定所返回的数组元素的最大数目。

说明

本函数返回由字符串组成的数组，其中的每个元素都是由 *separator* 作为边界点分割出来的子字符串。

separator 参数不能是空字符串。如果 *separator* 为空字符串（""），explode() 将返回 FALSE。如果 *separator* 所包含的值在 *string* 中找不到，那么 explode() 将返回包含 *string* 中单个元素的数组。

如果设置了 *limit* 参数，则返回的数组包含最多 *limit* 个元素，而最后那个元素将包含 *string* 的剩余部分。

如果 *limit* 参数是负数，则返回除了最后的 -*limit* 个元素外的所有元素。此特性是 PHP 5.1.0 中新增的。

提示和注释

注释：参数 limit 是在 PHP 4.0.1 中加入的。

注释：由于历史原因，虽然 implode() 可以接收两种参数顺序，但是 explode() 不行。你必须保证 *separator* 参数在 *string* 参数之前才行。

例子

在本例中，我们将把字符串分割为数组：

```
<?php
$str = "Hello world. It's a beautiful day.";
print_r (explode(" ",$str));
?>
```

输出：

```
Array
(
[0] => Hello
[1] => world.
[2] => It's
[3] => a
[4] => beautiful
[5] => day.
)
```

PHP fprintf() 函数

定义和用法

fprintf() 函数把格式化的字符串写到指定的输出流（例如：文件或数据库）。

该函数返回被写字符串的长度。

语法

```
fprintf(stream,format,arg1,arg2,arg++)
```

参数	描述
stream	可选。规定在哪里写/输出字符串。
format	必需。转换格式。
arg1	必需。规定插到 format 字符串中第一个 % 符号处的参数。
arg2	可选。规定插到 format 字符串中第二个 % 符号处的参数。
arg++	可选。规定插到 format 字符串中第三、四等等 % 符号处的参数。

说明

参数 *format* 是转换的格式，以百分比符号 ("%") 开始到转换字符结束。下面的可能的 *format* 值：

- %% - 返回百分比符号
- %b - 二进制数
- %c - 依照 ASCII 值的字符
- %d - 带符号十进制数
- %e - 可续计数法（比如 1.5e+3）
- %u - 无符号十进制数
- %f - 浮点数(local settings aware)
- %F - 浮点数(not local settings aware)
- %o - 八进制数
- %s - 字符串
- %x - 十六进制数（小写字母）
- %X - 十六进制数（大写字母）

arg1, arg2, ++ 等参数将插入到主字符串中的百分号 (%) 符号处。该函数是逐步执行的。在第一个 % 符号中，插入 arg1，在第二个 % 符号处，插入 arg2，依此类推。

提示和注释

注释：如果 % 符号多于 arg 参数，则您必须使用占位符。占位符被插入 % 符号之后，由数字和 "$" 组成。请参见例子 3。

提示： 相关函数： printf()、 sprintf()、 vfprintf()、 vprintf() 以及 vsprintf()。

例子

例子 1

```
<?php
$str = "Hello";
$number = 123;
$file = fopen("test.txt","w");
echo fprintf($file,"%s world. Day number %u",$str,$number);
?>
```

输出：

```
27
```

以下文本将写入 "test.txt"：

```
Hello world. Day number 123
```

例子 2

```
<?php
$number = 123;
$file = fopen("test.txt","w");
fprintf($file,"%f",$number);
?>
```

输出：

```
123.000000
```

例子 3

使用占位符：

```
<?php
$number = 123;
$file = fopen("test.txt","w");
fprintf($file,"With 2 decimals: %1\$.2f\nWith no decimals: %1\$u",$number);
?>
```

以下文本将写入 "test.txt"：

```
With 2 decimals: 123.00
With no decimals: 123
```

PHP get_html_translation_table() 函数

定义和用法

get_html_translation_table() 函数返回被 htmlentities() 和 htmlspecialchars() 函数使用的翻译表。

语法

```
get_html_translation_table(function,quotestyle)
```

参数	描述
function	可选。规定返回哪个翻译表。默认是 HTML_SPECIALCHARS。可能的值：`HTML_ENTITIES` - 翻译所有需要 URL 编码的字符，以便正确地显示在网页上。 `HTML_SPECIALCHARS` - 翻译某些需要 URL 编码的字符，以便正确地显示在网页上。
salt	可选。定义如何对单引号和双引号进行编码。默认是 ENT_COMPAT。可能的值： `ENT_COMPAT` - 编码双引号，不编码单引号。 `ENT_QUOTES` - 编码双引号和单引号。 `ENT_NOQUOTES` - 不编码单引号或双引号。

说明

提示和注释

提示：一些字符可以按照若干种方式进行编码。get_html_translation_table() 返回最普通的编码。

例子

在本例中，我们将展示两种翻译表：

```
<?php
print_r (get_html_translation_table());
echo "<br />";
print_r (get_html_translation_table(HTML_ENTITIES));
?>
```

输出：

```
Array
(
["] => " [<] => < [>] => > [&] => &
)
Array
(
[ ] =>   [¡] => ¡ [¢] => ¢ [£] => £
[¤] => ¤ [¥] => ¥ [¦] => ¦ [§] => §
[¨] => ¨ [©] => © [ª] => ª [«] => «
[¬] => ¬ [] =>  [®] => ® [¯] => ¯
[°] => ° [±] => ± [²] => ² [³] => ³
[´] => ´ [µ] => µ [¶] => ¶ [·] => ·
[¸] => ¸ [¹] => ¹ [º] => º [»] => »
[¼] => ¼ [½] => ½ [¾] => ¾ [¿] => ¿
[À] => À [Á] => Á [Â] => Â [Ã] => Ã
[Ä] => Ä [Å] => Å [Æ] => Æ [Ç] => Ç
[È] => È [É] => É [Ê] => Ê [Ë] => Ë
[Ì] => Ì [Í] => Í [Î] => Î [Ï] => Ï
[Ð] => Ð [Ñ] => Ñ [Ò] => Ò [Ó] => Ó
[Ô] => Ô [Õ] => Õ [Ö] => Ö [×] => ×
[Ø] => Ø [Ù] => Ù [Ú] => Ú [Û] => Û
[Ü] => Ü [Ý] => Ý [Þ] => Þ [ß] => ß
[à] => à [á] => á [â] => â [ã] => ã
[ä] => ä [å] => å [æ] => æ [ç] => ç
[è] => è [é] => é [ê] => ê [ë] => ë
[ì] => ì [í] => í [î] => î [ï] => ï
[ð] => ð [ñ] => ñ [ò] => ò [ó] => ó
[ô] => ô [õ] => õ [ö] => ö [÷] => ÷
[ø] => ø [ù] => ù [ú] => ú [û] => û
[ü] => ü [ý] => ý [þ] => þ [ÿ] => ÿ
["] => " [<] => < [>] => > [&] => &
)
```

PHP hebrev() 函数

定义和用法

hebrev() 函数把希伯来文本从右至左的流转换为左至右的流。

只有 224 至 251 之间的 ASCII 字符，以及标点符号受到影响。

语法

```
hebrev(string,maxcharline)
```

参数	描述
string	必需。希伯来文本。
salt	规定每行的最大字符数。如果可能，hebrev() 将避免把单词断开。

提示和注释

提示：hebrev() 和 hebrevc() 可以把希伯来逻辑文本转换为希伯来可见文本。希伯来可见文本不需要特殊的右至左字符支持，这使它对于在 web 上显示希伯来文本很有用处。

PHP hebrevc() 函数

定义和用法

hebrevc() 函数把希伯来文本从右至左的流转换为左至右的流。它也会把新行 (\n) 转换为
。

只有 224 至 251 之间的 ASCII 字符，以及标点符号受到影响。

语法

```
hebrev(string,maxcharline)
```

参数	描述
string	必需。希伯来文本。
salt	规定每行的最大字符数。如果可能，hebrev() 将避免把单词断开。

提示和注释

提示：hebrev() 和 hebrevc() 可以把希伯来逻辑文本转换为希伯来可见文本。希伯来可见文本不需要特殊的右至左字符支持，这使它对于在 web 上显示希伯来文本很有用处。

PHP html_entity_decode() 函数

定义和用法

html_entity_decode() 函数把 HTML 实体转换为字符。

html_entity_decode() 是 htmlentities() 的反函数。

语法

```
html_entity_decode(string,quotestyle,character-set)
```

参数	描述
string	必需。规定要解码的字符串。
quotestyle	可选。规定如何解码单引号和双引号。 ENT_COMPAT - 默认。仅解码双引号。 ENT_QUOTES - 解码双引号和单引号。 ENT_NOQUOTES - 不解码任何引号。
character-set	可选。字符串值，规定要使用的字符集。 ISO-8859-1 - 默认。西欧。 ISO-8859-15 - 西欧（增加 Euro 符号以及法语、芬兰语字母）。 UTF-8 - ASCII 兼容多字节 8 比特 Unicode cp866 - DOS 专用 Cyrillic 字符集 cp1251 - Windows 专用 Cyrillic 字符集 cp1252 - Windows 专用西欧字符集 KOI8-R - 俄语 GB2312 - 简体中文，国家标准字符集 BIG5 - 繁体中文 BIG5-HKSCS - Big5 香港扩展 Shift_JIS - 日语 EUC-JP - 日语

提示和注释

提示：无法被识别的字符集将被忽略，并由 ISO-8859-1 代替。

例子

```
<?php
$str = "John &amp; &#039;Adams&#039;";
echo html_entity_decode($str);
echo "<br />";
echo html_entity_decode($str, ENT_QUOTES);
echo "<br />";
echo html_entity_decode($str, ENT_NOQUOTES);
?>
```

浏览器输出：

```
John & 'Adams'
John & 'Adams'
John & 'Adams'
```

如果在浏览器中查看源代码，会看到这些 HTML：

```
<html>
<body>
John & &#039;Adams&#039;<br />
John & 'Adams'<br />
John & &#039;Adams&#039;
</body>
</html>
```

PHP htmlentities() 函数

定义和用法

htmlentities() 函数把字符转换为 HTML 实体。

语法

```
htmlentities(string,quotestyle,character-set)
```

参数	描述
string	必需。规定要转换的字符串。
quotestyle	可选。规定如何编码单引号和双引号。 `ENT_COMPAT` - 默认。仅编码双引号。 `ENT_QUOTES` - 编码双引号和单引号。 `ENT_NOQUOTES` - 不编码任何引号。
character-set	可选。字符串值，规定要使用的字符集。 `ISO-8859-1` - 默认。西欧。 `ISO-8859-15` - 西欧（增加 Euro 符号以及法语、芬兰语字母）。 `UTF-8` - ASCII 兼容多字节 8 比特 Unicode `cp866` - DOS 专用 Cyrillic 字符集 `cp1251` - Windows 专用 Cyrillic 字符集 `cp1252` - Windows 专用西欧字符集 `KOI8-R` - 俄语 `GB2312` - 简体中文，国家标准字符集 `BIG5` - 繁体中文 `BIG5-HKSCS` - Big5 香港扩展 `Shift_JIS` - 日语 `EUC` -JP - 日语

提示和注释

提示：无法被识别的字符集将被忽略，并由 ISO-8859-1 代替。

例子

```
<html>
<body>
<?php
$str = "John & 'Adams'";
echo htmlentities($str, ENT_COMPAT);
echo "<br />";
echo htmlentities($str, ENT_QUOTES);
echo "<br />";
echo htmlentities($str, ENT_NOQUOTES);
?>
</body>
</html>
```

浏览器输出：

```
John & 'Adams'
John & 'Adams'
John & 'Adams'
```

如果在浏览器中查看源代码，会看到这些 HTML：

```
<html>
<body>
John &amp; 'Adams'<br />
John &amp; &#039;Adams&#039;<br />
John &amp; 'Adams'
</body>
</html>
```

PHP htmlspecialchars_decode() 函数

定义和用法

htmlspecialchars_decode() 函数把一些预定义的 HTML 实体转换为字符。

会被解码的 HTML 实体是：

- & 成为 & （和号）
- " 成为 " （双引号）
- ' 成为 ' （单引号）
- < 成为 < （小于）
- > 成为 > （大于）

语法

```
htmlspecialchars_decode(string,quotestyle)
```

参数	描述
string	必需。规定要解码的字符串。
quotestyle	可选。规定如何解码单引号和双引号。 ENT_COMPAT - 默认。仅解码双引号。 ENT_QUOTES - 解码双引号和单引号。 ENT_NOQUOTES - 不解码任何引号。

例子

```
<?php
$str = "John &amp; &#039;Adams&#039;";
echo htmlspecialchars_decode($str);
echo "<br />";
echo htmlspecialchars_decode($str, ENT_QUOTES);
echo "<br />";
echo htmlspecialchars_decode($str, ENT_NOQUOTES);
?>
```

浏览器输出：

```
John & 'Adams'
John & 'Adams'
John & 'Adams'
```

如果在浏览器中查看源代码，会看到这些 HTML：

```
<html>
<body>
John & &#039;Adams&#039;<br />
John & 'Adams'<br />
John & &#039;Adams&#039;
</body>
</html>
```

PHP htmlspecialchars() 函数

定义和用法

htmlspecialchars() 函数把一些预定义的字符转换为 HTML 实体。

预定义的字符是：

- & （和号） 成为 &
- " （双引号） 成为 "
- ' （单引号） 成为 '
- < （小于） 成为 <
- > （大于） 成为 >

语法

```
htmlspecialchars(string,quotestyle,character-set)
```

参数	描述
string	必需。规定要转换的字符串。
quotestyle	可选。规定如何编码单引号和双引号。 `ENT_COMPAT` - 默认。仅编码双引号。 `ENT_QUOTES` - 编码双引号和单引号。 `ENT_NOQUOTES` - 不编码任何引号。
character-set	可选。字符串值，规定要使用的字符集。 `ISO-8859-1` - 默认。西欧。 `ISO-8859-15` - 西欧（增加 Euro 符号以及法语、芬兰语字母）。 `UTF-8` - ASCII 兼容多字节 8 比特 Unicode `cp866` - DOS 专用 Cyrillic 字符集 `cp1251` - Windows 专用 Cyrillic 字符集 `cp1252` - Windows 专用西欧字符集 `KOI8-R` - 俄语 `GB2312` - 简体中文，国家标准字符集 `BIG5` - 繁体中文 `BIG5-HKSCS` - Big5 香港扩展 `Shift_JIS` - 日语 `EUC-JP` - 日语

提示和注释

提示：无法被识别的字符集将被忽略，并由 ISO-8859-1 代替。

例子

```
<html>
<body>
<?php
$str = "John & 'Adams'";
echo htmlspecialchars($str, ENT_COMPAT);
echo "<br />";
echo htmlspecialchars($str, ENT_QUOTES);
echo "<br />";
echo htmlspecialchars($str, ENT_NOQUOTES);
?>
</body>
</html>
```

浏览器输出：

```
John & 'Adams'
John & 'Adams'
John & 'Adams'
```

如果在浏览器中查看源代码，会看到这些 HTML：

```
<html>
<body>
John &amp; 'Adams'<br />
John &amp; &#039;Adams&#039;<br />
John &amp; 'Adams'
</body>
</html>
```

PHP implode() 函数

定义和用法

implode() 函数把数组元素组合为一个字符串。

语法

```
implode(separator,array)
```

参数	描述
separator	可选。规定数组元素之间放置的内容。默认是 ""（空字符串）。
array	必需。要结合为字符串的数组。

说明

虽然 *separator* 参数是可选的。但是为了向后兼容，推荐您使用使用两个参数。

提示和注释

注释：implode() 可以接收两种参数顺序。但是由于历史原因，explode() 是不行的。你必须保证 *separator* 参数在 *string* 参数之前才行。

例子

```
<?php
$arr = array('Hello','World!','Beautiful','Day!');
echo implode(" ",$arr);
?>
```

输出：

```
Hello World! Beautiful Day!
```

PHP join() 函数

定义和用法

join() 函数把数组元素组合为一个字符串。

join() 函数是 implode() 函数的别名。

语法

```
join(separator,array)
```

参数	描述
separator	可选。规定数组元素之间放置的内容。默认是 ""（空字符串）。
array	必需。要结合为字符串的数组。

说明

虽然 *separator* 参数是可选的。但是为了向后兼容，推荐您使用使用两个参数。

提示和注释

注释：join() 可以接收两种参数顺序。但是由于历史原因，explode() 是不行的。你必须保证 *separator* 参数在 *string* 参数之前才行。

例子

```
<?php
$arr = array('Hello','World!','Beautiful','Day!');
echo join(" ",$arr);
?>
```

输出：

```
Hello World! Beautiful Day!
```

PHP levenshtein() 函数

定义和用法

levenshtein() 函数返回两个字符串之间的 Levenshtein 距离。

Levenshtein 距离，又称编辑距离，指的是两个字符串之间，由一个转换成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。

例如把 kitten 转换为 sitting：

1. sitten （k→s）
2. sittin （e→i）
3. sitting （→g）

levenshtein() 函数给每个操作（替换、插入和删除）相同的权重。不过，您可以通过设置可选的 insert、replace、delete 参数，来定义每个操作的代价。

语法

```
levenshtein(string1,string2,insert,replace,delete)
```

参数	描述
string1	必需。要对比的第一个字符串。
string2	必需。要对比的第二个字符串。
insert	可选。插入一个字符的代价。默认是 1。
replace	可选。替换一个字符的代价。默认是 1。
delete	可选。删除一个字符的代价。默认是 1。

提示和注释

注释：如果其中一个字符串超过 255 个字符，levenshtein() 函数返回 -1。

注释：levenshtein() 函数对大小写不敏感。

注释：levenshtein() 函数比 similar_text() 函数更快。不过，similar_text() 函数提供需要更少修改的更精确的结果。

例子

```
<?php
echo levenshtein("Hello World","ello World");
echo "<br />";
echo levenshtein("Hello World","ello World",10,20,30);
?>
```

输出：

```
1
30
```

PHP localeconv() 函数

定义和用法

localeconv() 函数返回包含本地数字及货币信息格式的数组。

localeconv() 函数返回以下数组元素：

- [decimal_point] - 小数点字符
- [thousands_sep] - 千位分隔符
- [int_curr_symbol] - 货币符号 （例如：USD）
- [currency_symbol] - 货币符号 （例如：$）
- [mon_decimal_point] - 货币小数点符号
- [mon_thousands_sep] - 货币千位分隔符
- [positive_sign] - 正值符号
- [negative_sign] - 负值符号
- [int_frac_digits] - 国际小数数字
- [frac_digits] - 本地小数数字
- [p_cs_precedes] - if 如果货币符号在正值之前，则是 True (1)，否则是 False (0)。
- [p_sep_by_space] - True (1) 如果货币符号与正值之间有空间，则是 True (1)，否则是 False (0)。
- [n_cs_precedes] - True (1) if 货币符号在负值之前，则是 True (1)，否则是 False (0)。
- [p_sep_by_space] - True (1) 如果货币符号与负值之间有空间，则是 True (1)，否则是 False (0)。
- [p_sign_posn] - 格式化选项：
  - 0 - 在数量和货币符号周围的圆括号
  - 1 - 数量和货币符号之前的 + 号
  - 2 - 数量和货币符号之后的 + 号
  - 3 - 货币符号之前的 + 号
  - 4 - 货币符号之后的 + 号
- [n_sign_posn] - 格式化选项：
  - 0 - 在数量和货币符号周围的圆括号
  - 1 - 数量和货币符号之前的 - 号
  - 2 - 数量和货币符号之后的 - 号
  - 3 - 货币符号之前的 - 号
  - 4 - 货币符号之后的 - 号
- [grouping] - 显示如何分组数字的 Array（例如：3 指示 1 000 000）
- [mon_grouping] - 显示如何分组货币数字的 Array（例如：2 指示 1 00 00 00）

语法

```
localeconv()
```

提示和注释

提示：如需定义本地设置，请使用 setlocale() 函数。

例子

在本例中，我们将获得美国本地的数字格式化信息：

```
<?php
setlocale(LC_ALL, 'US');
$locale_info = localeconv();
print_r($locale_info);
?>
```

输出：

```
Array
(
[decimal_point] => .
[thousands_sep] => ,
[int_curr_symbol] => USD
[currency_symbol] => $
[mon_decimal_point] => .
[mon_thousands_sep] => ,
[positive_sign] =>
[negative_sign] => -
[int_frac_digits] => 2
[frac_digits] => 2
[p_cs_precedes] => 1
[p_sep_by_space] => 0
[n_cs_precedes] => 1
[n_sep_by_space] => 0
[p_sign_posn] => 3
[n_sign_posn] => 0
[grouping] => Array ([0] => 3)
[mon_grouping] => Array ([0] => 3)
)
```

PHP ltrim() 函数

定义和用法

ltrim() 函数从字符串左侧删除空格或其他预定义字符。

语法

```
ltrim(string,charlist)
```

参数	描述
string	必需。规定要转换的字符串。
charlist	可选。规定从字符串中删除哪些字符。如果未设置该参数，则全部删除以下字符： " \0 " - ASCII 0, NULL " \t " - ASCII 9, 制表符 " \n " - ASCII 10, 新行 " \x0B " - ASCII 11, 垂直制表符 " \r " - ASCII 13, 回车 " " - ASCII 32, 空格

例子

例子 1

```
<html>
<body>
<?php
$str = "    Hello World!";
echo "Without ltrim: " . $str;
echo "<br />";
echo "With ltrim: " . ltrim($str);
?>
<body>
<html>
```

输出：

```
Without ltrim: Hello World!
With ltrim: Hello World!
```

如果在浏览器中查看源代码，会看到以下 HTML：

```
<html>
<body>
Without ltrim:     Hello World!<br />With ltrim: Hello World!
</body>
</html>
```

例子 2

```
<?php
$str = "\r\nHello World!";
echo "Without ltrim: " . $str;
echo "<br />";
echo "With ltrim: " . ltrim($str);
?>
```

输出：

```
Without ltrim: Hello World!
With ltrim: Hello World!
```

如果在浏览器中查看源代码，会看到以下 HTML：

```
<html>
<body>
Without ltrim:
Hello World!<br />With ltrim: Hello World!
</body>
</html>
```

PHP md5() 函数

定义和用法

md5() 函数计算字符串的 MD5 散列。

md5() 函数使用 RSA 数据安全，包括 MD5 报文摘译算法。

如果成功，则返回所计算的 MD5 散列，如果失败，则返回 false。

语法

```
md5(_string_,_raw_)
```

参数	描述
string	必需。规定要计算的字符串。
raw	可选。规定十六进制或二进制输出格式： `TRUE` - 原始 16 字符二进制格式 `FALSE` - 默认。32 字符十六进制数 注释：该参数是 PHP 5.0 中添加的。

例子

例子 1

```
<?php
$str = "Hello";
echo md5($str);
?>
```

输出：

```
8b1a9953c4611296a827abf8c47804d7
```

例子 2

```
<?php
$str = "Hello";
echo md5($str);

if (md5($str) == '8b1a9953c4611296a827abf8c47804d7')
  {
  echo "<br />Hello world!";
  exit;
  }
?>
```

输出：

```
8b1a9953c4611296a827abf8c47804d7
Hello world!
```

PHP md5_file() 函数

定义和用法

md5_file() 函数计算文件的 MD5 散列。

md5() 函数使用 RSA 数据安全，包括 MD5 报文摘译算法。

如果成功，则返回所计算的 MD5 散列，如果失败，则返回 false。

语法

```
md5(_string_,_raw_)
```

参数	描述
string	必需。规定要计算的文件。
raw	可选。规定十六进制或二进制输出格式： `TRUE` - 原始 16 字符二进制格式 `FALSE` - 默认。32 字符十六进制数注释：该参数是 PHP 5.0 中添加的。

例子

例子 1

```
<?php
$filename = "test.txt";
$md5file = md5_file($filename);
echo $md5file;
?>
```

输出：

```
5d41402abc4b2a76b9719d911017c592
```

例子 2

存储 "test.txt" 文件的 MD5 散列：

```
<?php
$md5file = md5_file("test.txt");
file_put_contents("md5file.txt",$md5file);
?>
```

在本例中，我们将检测 "test.txt" 是否已被更改（即是否 MD5 散列已被更改）：

```
<?php
$md5file = file_get_contents("md5file.txt");
if (md5_file("test.txt") == $md5file)
  {
  echo "The file is ok.";
  }
else
  {
  echo "The file has been changed.";
  }
?>
```

输出：

```
The file is ok.
```

定义和用法

metaphone() 函数计算字符串的 metaphone 键。

metaphone 键字符串的英语发音。

metaphone() 函数可用于拼写检查应用程序。

如果成功，则返回字符串的 metaphone 键，如果失败，则返回 false。

语法

```
metaphone(string,length)
```

参数	描述
string	必需。规定要检查的字符串。
length	可选。规定 metaphone 键的最大长度。

提示和注释

注释：metaphone() 为发音相似的单词创建相同的键。

注释：所生成的 metaphone 键长度可变。

提示：metaphone() 比 soundex() 函数更精确，因为 metaphone() 了解基本的英语发音规则。

例子

例子 1

```
<?php
echo metaphone("world");
?>
```

输出：

```
WRLT
```

例子 2

在本例中，我们对两个发音相似的单词应用 metaphone() 函数：

```
<?php
$str = "Sun";
$str2 = "Son";

echo metaphone($str);
echo metaphone($str2);
?>
```

输出：

```
SN
SN
```

PHP money_format() 函数

定义和用法

money_format() 函数把字符串格式化为货币字符串。

语法

```
money_format(string,number)
```

参数	描述
string	必需。规定要格式化的字符串。
number	可选。被插入格式化字符串中 % 符号位置的数字。

提示和注释

注释：money_format() 函数无法在 windows 平台上工作。

例子

例子 1

国际 en_US 格式：

```
<?php
$number = 1234.56;
setlocale(LC_MONETARY, "en_US");
echo money_format("The price is %i", $number);
?>
```

输出：

```
The price is USD 1,234.56
```

例子 2

负数，带有 () 指示负数的 US 国际格式，右侧精度为 2，"*" 为填充字符：

```
<?php
$number = -1234.5672;

echo money_format("%=*(#10.2n", $number);
?>
```

输出：

```
($********1,234.57)
```

PHP nl_langinfo() 函数

定义和用法

nl_langinfo() 函数返回指定的本地信息。

如果成功，则返回指定的本地信息。如果失败，则返回 false。

语法

```
nl_langinfo(element)
```

参数	描述
element	必需。规定要返回哪个元素。必须是说明中列出的元素之一。

说明

时间和日历：

- ABDAY_(1-7) - Abbreviated name of the numbered day of the week
- DAY_(1-7) - Name of the numbered day of the week (DAY_1 = Sunday)
- ABMON_(1-12) - Abbreviated name of the numbered month of the year
- MON_(1-12) - Name of the numbered month of the year
- AM_STR - String for Ante meridian
- PM_STR - String for Post meridian
- D_T_FMT - String that can be used as the format string for strftime() to represent time and date
- D_FMT - String that can be used as the format string for strftime() to represent date
- T_FMT - String that can be used as the format string for strftime() to represent time
- T_FMT_AMPM - String that can be used as the format string for strftime() to represent time in 12-hour format with ante/post meridian
- ERA - Alternate era
- ERA_YEAR - Year in alternate era format
- ERA_D_T_FMT - Date and time in alternate era format (string can be used in strftime())
- ERA_D_FMT - Date in alternate era format (string can be used in strftime())
- ERA_T_FMT - Time in alternate era format (string can be used in strftime())

货币类别：

- INT_CURR_SYMBOL - Currency symbol (example: USD)
- CURRENCY_SYMBOL - Currency symbol (example: $)
- CRNCYSTR - Same as CURRENCY_SYMBOL
- MON_DECIMAL_POINT - Monetary decimal point character
- MON_THOUSANDS_SEP - Monetary thousands separator
- POSITIVE_SIGN - Positive value character
- NEGATIVE_SIGN -Negative value character
- MON_GROUPING - Array displaying how monetary numbers are grouped (example: 1 000 000)
- INT_FRAC_DIGITS - International fractional digits
- FRAC_DIGITS - Local fractional digits
- P_CS_PRECEDES - True (1) if currency symbol is placed in front of a positive value, False (0) if it is placed behind
- P_SEP_BY_SPACE - True (1) if there is a spaces between the currency symbol and a positive value, False (0) otherwise
- N_CS_PRECEDES - True (1) if currency symbol is placed in front of a negative value, False (0) if it is placed behind
- N_SEP_BY_SPACE - True (1) if there is a spaces between the currency symbol and a negative value, False (0) otherwise
- P_SIGN_POSN - Formatting setting. Possible return values:
  - 0 - Parentheses surround the quantity and currency symbol
  - 1 - The sign string is placed in front of the quantity and currency symbol
  - 2 - The sign string is placed after the quantity and currency symbol
  - 3 - The sign string is placed immediately in front of the currency symbol
  - 4 - The sign string is placed immediately after the currency symbol
- N_SIGN_POSN - Formatting setting. Possible return values:
  - 0 - Parentheses surround the quantity and currency symbol
  - 1 - The sign string is placed in front of the quantity and currency symbol
  - 2 - The sign string is placed after the quantity and currency symbol
  - 3 - The sign string is placed immediately in front of the currency symbol
  - 4 - The sign string is placed immediately after the currency symbol

数字类别：

- DECIMAL_POINT - Decimal point character
- RADIXCHAR - Same as DECIMAL_POINT
- THOUSANDS_SEP - Separator character for thousands
- THOUSEP - Same as THOUSANDS_SEP
- GROUPING - Array displaying how numbers are grouped (example: 1 000 000)

通信类别：

- YESEXPR - Regex string for matching 'yes' input
- NOEXPR - Regex string for matching 'no' input
- YESSTR - Output string for 'yes'
- NOSTR - Output string for 'no'

代码集类别：

- CODESET Return a string with the name of the character encoding.

提示和注释

注释：money_format() 函数无法在 windows 平台上工作。

提示：与返回所有本地格式化信息的 localeconv() 函数不同，nl_langinfo() 返回指定的信息。

PHP nl2br() 函数

定义和用法

nl2br() 函数在字符串中的每个新行 (\n) 之前插入 HTML 换行符 (<br />)。

语法

```
nl2br(string)
```

参数	描述
string	必需。规定要检查的字符串。

例子

```
<?php
echo nl2br("One line.\nAnother line.");
?>
```

输出：

```
One line.
Another line.
```

HTML 代码：

```
One line.<br />
Another line.
```

PHP number_format() 函数

定义和用法

number_format() 函数通过千位分组来格式化数字。

语法

```
number_format(number,decimals,decimalpoint,separator)
```

参数	描述
number	必需。要格式化的数字。如果未设置其他参数，则数字会被格式化为不带小数点且以逗号 (,) 作为分隔符。
decimals	可选。规定多少个小数。如果设置了该参数，则使用点号 (.) 作为小数点来格式化数字。
decimalpoint	可选。规定用作小数点的字符串。
separator	可选。规定用作千位分隔符的字符串。仅使用该参数的第一个字符。比如 "xyz" 仅输出 "x"。注释：如果设置了该参数，那么所有其他参数都是必需的。

提示和注释

注释：该函数支持一个、两个或四个参数（不是三个）。

例子

```
<?php
echo number_format("1000000");
echo number_format("1000000",2);
echo number_format("1000000",2,",",".");
?>
```

输出：

```
1,000,000
1,000,000.00
1.000.000,00
```

PHP ord() 函数

定义和用法

ord() 函数返回字符串第一个字符的 ASCII 值。

语法

```
ord(string)
```

参数	描述
string	必需。要从中获得 ASCII 值的字符串。

例子

```
<?php
echo ord("h");
echo ord("hello");
?>
```

输出：

```
104
104
```

PHP parse_str() 函数

定义和用法

parse_str() 函数把查询字符串解析到变量中。

语法

```
parse_str(string,array)
```

参数	描述
string	必需。规定要解析的字符串。
array	可选。规定存储变量的数组名称。该参数指示变量存储到数组中。

提示和注释

注释：如果未设置 array 参数，由该函数设置的变量将覆盖已由同名变量。

注释：php.ini 中的 magic_quotes_gpc 设置影响该函数的输出。如果已启用，那么在 parse_str() 解析之前，变量会被 addslashes() 转换。

例子

例子 1

```
<?php
parse_str("id=23&name=John%20Adams");
echo $id."<br />";
echo $name;
?>
```

输出：

```
23
John Adams
```

例子 2

```
<?php
parse_str("id=23&name=John%20Adams",$myArray);
print_r($myArray);
?>
```

输出：

```
Array
(
[id] => 23
[name] => John Adams
)
```

PHP print() 函数

定义和用法

print() 函数输出一个或多个字符串。

语法

```
print(strings)
```

参数	描述
strings	必需。发送到输出的一个或多个字符串。

提示和注释

注释：print() 函数实际上不是函数，所以您不必对它使用括号。

注释：print() 函数稍慢于 echo()。

例子

例子 1

```
<?php
$str = "Who's John Adams?";
print $str;
print "<br />";
print $str."<br />I don't know!";
?>
```

输出：

```
Who's John Adams?
Who's John Adams?
I don't know!
```

例子 2

```
<?php
print "This text
spans multiple
lines.";
?>
```

输出：

```
This text spans multiple lines.
```

例子 3

```
<?php
$color = "red";
print "Roses are $color";
print "<br />";
print 'Roses are $color';
?>
```

输出：

```
Roses are red
Roses are $color
```

PHP printf() 函数

定义和用法

printf() 函数输出格式化的字符串。

语法

```
printf(format,arg1,arg2,arg++)
```

参数	描述
format	必需。规定字符串以及如何格式化其中的变量。
arg1	必需。规定插到格式化字符串中第一个 % 符号处的参数。
arg2	可选。规定插到格式化字符串中第二个 % 符号处的参数。
arg++	可选。规定插到格式化字符串中第三、四等等 % 符号处的参数。

说明

arg1, arg2, ++ 等参数将插入到主字符串中的百分号 (%) 符号处。该函数是逐步执行的。在第一个 % 符号中，插入 arg1，在第二个 % 符号处，插入 arg2，依此类推。

提示和注释

注释：如果 % 符号多于 arg 参数，则您必须使用占位符。占位符被插入 % 符号之后，由数字和 "$" 组成。请参见例子 3。

注释： 相关函数： fprintf()、 sprintf()、 vfprintf()、 vprintf() 以及 vsprintf()。

例子

例子 1

```
<?php
$str = "Hello";
$number = 123;
printf("%s world. Day number %u",$str,$number);
?>
```

输出：

```
Hello world. Day number 123
```

例子 2

```
<?php
$number = 123;
printf("%f",$number);
?>
```

输出：

```
123.000000
```

例子 3

使用占位符：

```
<?php
$number = 123;
printf("With 2 decimals: %1\$.2f<br />With no decimals: %1\$u",$number);
?>
```

输出：

```
With 2 decimals: 123.00
With no decimals: 123
```

PHP quoted_printable_decode() 函数

定义和用法

quoted_printable_decode() 函数对经过 quoted-printable 编码后的字符串进行解码，返回 8 位的字符串。

该函数类似于 imap_qprint() 函数。不同的是，应用 imap_qprint() 函数需要让系统加载 IMAP 模块，而本函数不需要加载 IMAP 模块。

语法

```
quoted_printable_decode(string)
```

参数	描述
string	必需。规定要解码的 quoted-printable 字符串。

例子

在本例中，应用 quoted_printable_decode() 函数对经过 quoted-printable 编码后的字符串 "=0A" 进行解码，并返回 8 位的字符串：

```
<?php
$str = "Hello=0Aworld.";
echo quoted_printable_decode($str);
?>
```

输出：

```
Hello world.
```

HTML 源代码：

```
Hello
world.
```

PHP quotemeta() 函数

定义和用法

quotemeta() 函数在字符串中某些预定义的字符前添加反斜杠。

这些预定义字符是：

- 句号 (.)
- 反斜杠 ()
- 加号 (+)
- 星号 (*)
- 问号 (?)
- 方括号 ([])
- 脱字符号 (^)
- 美元符号 ($)
- 圆括号 (())

语法

```
quotemeta(string)
```

参数	描述
string	必需。规定要检查的字符串。

提示和注释

提示：该函数可用于转义拥有特殊意义的字符，比如 SQL 中的 ( )、[ ] 以及 *。

例子

```
<?php
$str = "Hello world. (can you hear me?)";
echo quotemeta($str);
?>
```

输出：

```
Hello world\. \(can you hear me\?\)
```

PHP rtrim() 函数

定义和用法

chop() 函数从字符串的末端开始删除空白字符或其他预定义字符。

语法

```
rtrim(string,charlist)
```

参数	描述
string	必需。规定要转换的字符串。
charlist	可选。规定从字符串中删除哪些字符。如果未设置该参数，则全部删除以下字符： " `\0` " - ASCII 0, NULL " `\t` " - ASCII 9, 制表符 " `\n` " - ASCII 10, 新行 " `\x0B` " - ASCII 11, 垂直制表符 " `\r` " - ASCII 13, 回车 " " - ASCII 32, 空格

例子

在本例中，我们将使用 rtrim() 函数从字符串右端删除字符：

```
<?php
$str = "Hello World!\n\n";
echo $str;
echo rtrim($str);
?>
```

以上代码输出的源代码：

```
<html>

<body>
Hello World!

Hello World!</body>

</html>
```

输出：

```
Hello World! Hello World!
```

PHP setlocale() 函数

定义和用法

setlocale() 函数设置地区信息（地域信息）。

地区信息是针对一个地理区域的语言、货币、时间以及其他信息。

该函数返回当前的地区设置，若失败则返回 false。

语法

```
setlocale(constant,location)
```

参数	描述
constant	必需。规定应该设置什么地区信息。可用的常量： `LC_ALL` - 包括下面的所有选项 `LC_COLLATE` -排序次序 `LC_CTYPE` - 字符类别及转换（例如所有字符大写或小写） `LC_MESSAGES` - 系统消息格式 `LC_MONETARY` - 货币格式 `LC_NUMERIC` - 数字格式 `LC_TIME` - 日期/时间格式
location	必需。规定把地区信息设置为什么国家/地区。如果 location 参数是数组，setlocale() 会尝试每个数组元素，直到找到合法的语言或地区代码为止。如果某个地区在不同的系统上拥有不同的名称，这一点很有用。注释：在此查找语言和地区代码。

提示和注释

注释：setlocale() 函数仅针对当前脚本改变地区信息。

提示：可以通过 setlocale(LC_ALL,NULL) 把地区信息设置为系统默认。

例子

在本例中，我们将把 locale 设置为 US English，然后再设置回系统默认：

```
<?php
echo setlocale(LC_ALL,"En-Us");
echo setlocale(LC_ALL,NULL);
?>
```

PHP sha1() 函数

定义和用法

sha1() 函数计算字符串的 SHA-1 散列。

sha1() 函数使用美国 Secure Hash 算法 1。

如果成功，则返回所计算的 SHA-1 散列，如果失败，则返回 false。

语法

```
sha1(_string_,_raw_)
```

参数	描述
string	必需。规定要计算的字符串。
raw	可选。规定十六进制或二进制输出格式： `TRUE` - 原始 20 字符二进制格式 `FALSE` - 默认。40 字符十六进制数注释：该参数是 PHP 5.0 中添加的。

例子

例子 1

```
<?php
$str = 'Hello';
echo sha1($str);
?>
```

输出：

```
f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
```

例子 2

在本例中，我们将输出 sha1() 的结果，然后对其测试：

```
<?php
$str = 'Hello';
echo sha1($str);

if (sha1($str) == 'f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0')
  {
  echo "<br />Hello world!";
  exit;
  }
?>
```

输出：

```
f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
Hello world!
```

PHP sha1_file() 函数

定义和用法

sha1_file() 函数计算文件的 SHA-1 散列。

sha1() 函数使用美国 Secure Hash 算法 1。

如果成功，则返回所计算的 SHA-1 散列，如果失败，则返回 false。

语法

```
sha1_file(_string_,_raw_)
```

参数	描述
string	必需。规定要计算的文件。
raw	可选。规定十六进制或二进制输出格式： `TRUE` - 原始 20 字符二进制格式 `FALSE` - 默认。40 字符十六进制数注释：该参数是 PHP 5.0 中添加的。

例子

例子 1

```
<?php
$filename = "test.txt";
$sha1file = sha1_file($filename);
echo $sha1file;
?>
```

输出：

```
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
```

例子 2

在一个文件中存储 "test.txt" 的 SHA-1 散列：

```
<?php
$sha1file = sha1_file("test.txt");
file_put_contents("sha1file.txt",$sha1file);
?>
```

在本例中，我们将测试 "test.txt" 是否已更改（即 SHA-1 hash 是否已更改）：

```
<?php
$sha1file = file_get_contents("sha1file.txt");
if (sha1_file("test.txt") == $sha1file)
  {
  echo "The file is ok.";
  }
else
  {
  echo "The file has been changed.";
  }
?>
```

输出：

```
The file is ok.
```

PHP similar_text() 函数

定义和用法

similar_text() 函数计算两个字符串的匹配字符的数目。

该函数也可以计算两个字符串的相似度（以百分比计）。

语法

```
similar_text(string1,string2,percent)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。
percent	可选。规定供存储百分比相似度的变量名。

提示和注释

注释：levenshtein() 函数比 similar_text() 函数更快。不过，similar_text() 函数通过更少的必需修改次数提供更精确的结果。

例子

例子 1

```
<?php
echo similar_text("Hello World","Hello Peter");
?>
```

输出：

```
7
```

例子 2

```
<?php
similar_text("Hello World","Hello Peter",$percent);
echo $percent;
?>
```

输出：

```
63.6363636364
```

PHP soundex() 函数

定义和用法

soundex() 函数计算字符串的 soundex 键。

soundex 键是 4 字符长的字母数字字符串，表示一个单词的英文发音。

soundex() 函数可用于拼写检查程序。

如果成功，则返回字符串的 soundex 键，如果失败，则返回 false。

语法

```
soundex(string)
```

参数	描述
string	必需。规定要检查的字符串。

提示和注释

注释：soundex() 为发音相似的单词创建相同的键。

提示：metaphone() 比 soundex() 函数更精确，因为 metaphone() 了解基本的英语发音规则。

例子

例子 1

```
<?php
$str = "hello";
echo soundex($str);
?>
```

输出：

```
H400
```

例子 2

在本例中，我们对两个发音相似的单词应用 soundex() 函数：

```
<?php
$str = "Sun";
$str2 = "Son";

echo soundex($str);
echo "<br />";
echo soundex($str2);
?>
```

输出：

```
S500
S500
```

PHP sprintf() 函数

定义和用法

sprintf() 函数把格式化的字符串写入一个变量中。

语法

```
sprintf(_format_,_arg1_,_arg2_,_arg++_)
```

参数	描述
format	必需。转换格式。
arg1	必需。规定插到 *format* 字符串中第一个 % 符号处的参数。
arg2	可选。规定插到 *format* 字符串中第二个 % 符号处的参数。
arg++	可选。规定插到 *format* 字符串中第三、四等等 % 符号处的参数。

说明

参数 *format* 是转换的格式，以百分比符号 ("%") 开始到转换字符结束。下面的可能的 *format* 值：

- %% - 返回百分比符号
- %b - 二进制数
- %c - 依照 ASCII 值的字符
- %d - 带符号十进制数
- %e - 可续计数法（比如 1.5e+3）
- %u - 无符号十进制数
- %f - 浮点数(local settings aware)
- %F - 浮点数(not local settings aware)
- %o - 八进制数
- %s - 字符串
- %x - 十六进制数（小写字母）
- %X - 十六进制数（大写字母）

arg1, *arg2*, ++ 等参数将插入到主字符串中的百分号 (%) 符号处。该函数是逐步执行的。在第一个 % 符号中，插入 *arg1*，在第二个 % 符号处，插入 *arg2*，依此类推。

提示和注释

注释：如果 % 符号多于 *arg* 参数，则您必须使用占位符。占位符插到 % 符号后面，由数字和 "$" 组成。请参见例子 3。

提示： 相关函数： fprintf()、 printf()、 vfprintf()、 vprintf() 以及 vsprintf()。

例子

例子 1

```
<?php
$str = "Hello";
$number = 123;
$txt = sprintf("%s world. Day number %u",$str,$number);
echo $txt;
?>
```

输出：

```
Hello world. Day number 123
```

例子 2

```
<?php
$number = 123;
$txt = sprintf("%f",$number);
echo $txt;
?>
```

输出：

```
123.000000
```

例子 3

```
<?php
$number = 123;
$txt = sprintf("With 2 decimals: %1\$.2f<br />With no decimals: %1\$u",$number);
echo $txt;
?>
```

输出：

```
With 2 decimals: 123.00
With no decimals: 123
```

PHP sscanf() 函数

定义和用法

sscanf() 函数根据指定的格式解析来自一个字符串的输入。

如果只向该函数传递两个参数，数据将以数组的形式返回。否则，如果传递了额外的参数，那么被解析的数据会存储在这些参数中。如果区分符的数目大于包含它们的变量的数目，则会发生错误。不过，如果区分符少于变量，则额外的变量包含 NULL。

语法

```
sscanf(string,format,arg1,arg2,arg++)
```

参数	描述
string	必需。规定要读取的字符串。
format	必需。规定要使用的格式。
arg1	可选。存储数据的第一个变量。
arg2	可选。存储数据的第二个变量。
arg++	可选。存储数据的第三、四个变量。依此类推。

说明

参数 *format* 是转换的格式，以百分比符号 ("%") 开始到转换字符结束。下面的可能的 *format* 值：

- %% - 返回百分比符号
- %b - 二进制数
- %c - 依照 ASCII 值的字符
- %d - 带符号十进制数
- %e - 可续计数法（比如 1.5e+3）
- %u - 无符号十进制数
- %f - 浮点数(local settings aware)
- %F - 浮点数(not local settings aware)
- %o - 八进制数
- %s - 字符串
- %x - 十六进制数（小写字母）

- %X - 十六进制数（大写字母）

例子

```
<?php
$string = "age:30 weight:60kg";
sscanf($string,"age:%d weight:%dkg",$age,$weight);
// show types and values
var_dump($age,$weight);
?>
```

输出：

```
int(30)
int(60)
```

PHP str_ireplace() 函数

定义和用法

str_ireplace() 函数使用一个字符串替换字符串中的另一些字符。

语法

```
str_ireplace(find,replace,string,count)
```

参数	描述
find	必需。规定要查找的值。
replace	必需。规定替换 *find* 中的值的值。
string	必需。规定被搜索的字符串。
count	可选。一个变量，对替换数进行计数。

提示和注释

注释：该函数对大小写不敏感。请使用 str_replace() 执行对大小写敏感的搜索。

注释：该函数是二进制安全的。

例子

例子 1

```
<?php
echo str_ireplace("world","John","Hello world!");
?>
```

输出：

```
Hello John!
```

例子 2

在本例中，我们将演示带有数组和 count 变量的 str_ireplace() 函数：

```
<?php
$arr = array("blue","red","green","yellow");
print_r(str_ireplace("red","pink",$arr,$i));
echo "Replacements: $i";
?>
```

输出：

```
Array
(
[0] => blue
[1] => pink
[2] => green
[3] => yellow
)
Replacements: 1
```

例子 3

```
<?php
$find = array("Hello","world");
$replace = array("B");
$arr = array("Hello","world","!");
print_r(str_ireplace($find,$replace,$arr));
?>
```

输出：

```
Array
(
[0] => B
[1] =>
[2] => !
)
```

PHP str_pad() 函数

定义和用法

str_pad() 函数把字符串填充为指定的长度。

语法

```
str_pad(string,length,pad_string,pad_type)
```

参数	描述
string	必需。规定要填充的字符串。
length	必需。规定新字符串的长度。如果该值小于原始字符串的长度，则不进行任何操作。
pad_string	可选。规定供填充使用的字符串。默认是空白。
pad_type	可选。规定填充字符串的那边。可能的值： `STR_PAD_BOTH` - 填充到字符串的两头。如果不是偶数，则右侧获得额外的填充。 `STR_PAD_LEFT` - 填充到字符串的左侧。 `STR_PAD_RIGHT` - 填充到字符串的右侧。这是默认的。

例子

例子 1

```
<?php
$str = "Hello World";
echo str_pad($str,20,".");
?>
```

输出：

```
Hello World.........
```

例子 2

```
<?php
$str = "Hello World";
echo str_pad($str,20,".",STR_PAD_LEFT);
?>
```

输出：

```
.........Hello World
```

例子 3

```
<?php
$str = "Hello World";
echo str_pad($str,20,".:",STR_PAD_BOTH);
?>
```

输出：

```
.:.:Hello World.:.:.
```

PHP str_repeat() 函数

定义和用法

str_repeat() 函数把字符串重复指定的次数。

语法

```
str_repeat(string,repeat)
```

参数	描述
string	必需。规定要重复的字符串。
repeat	必需。规定字符串将被重复的次数。必须大于等于 0。

例子

```
<?php
echo str_repeat(".",13);
?>
```

输出：

```
.............
```

PHP str_replace() 函数

定义和用法

str_replace() 函数使用一个字符串替换字符串中的另一些字符。

语法

```
str_replace(find,replace,string,count)
```

参数	描述
find	必需。规定要查找的值。
replace	必需。规定替换 *find* 中的值的值。
string	必需。规定被搜索的字符串。
count	可选。一个变量，对替换数进行计数。

提示和注释

注释：该函数对大小写敏感。请使用 str_ireplace() 执行对大小写不敏感的搜索。

注释：该函数是二进制安全的。

例子

例子 1

```
<?php
echo str_replace("world","John","Hello world!");
?>
```

输出：

```
Hello John!
```

例子 2

在本例中，我们将演示带有数组和 count 变量的 str_replace() 函数：

```
<?php
$arr = array("blue","red","green","yellow");
print_r(str_replace("red","pink",$arr,$i));
echo "Replacements: $i";
?>
```

输出：

```
Array
(
[0] => blue
[1] => pink
[2] => green
[3] => yellow
)
Replacements: 1
```

例子 3

```
<?php
$find = array("Hello","world");
$replace = array("B");
$arr = array("Hello","world","!");
print_r(str_replace($find,$replace,$arr));
?>
```

输出：

```
Array
(
[0] => B
[1] =>
[2] => !
)
```

PHP str_rot13() 函数

定义和用法

str_rot13() 函数对字符串执行 ROT13 编码。

ROT-13 编码是一种每一个字母被另一个字母代替的方法。这个代替字母是由原来的字母向前移动 13 个字母而得到的。数字和非字母字符保持不变。

语法

```
str_rot13(string)
```

参数	描述
string	必需。规定要编码的字符串。

提示和注释

提示：编码和解码都是由相同的函数完成的。如果您把一个已编码的字符串作为参数，那么将返回原始字符串。

例子

在本例中，我们将通过使用 str_rot13() 函数对字符串进行编码和解码：

```
<?php
echo str_rot13("Hello World");
echo "<br />";
echo str_rot13("Uryyb Jbeyq");
?>
```

输出：

```
Uryyb Jbeyq
Hello World
```

PHP str_shuffle() 函数

定义和用法

str_shuffle() 函数随机地打乱字符串中的所有字符。

语法

```
str_shuffle(string)
```

参数	描述
string	必需。规定要打乱的字符串。

例子

```
<?php
echo str_shuffle("Hello World");
?>
```

输出：

```
H elooWlrdl
```

PHP str_split() 函数

定义和用法

str_split() 函数把字符串分割到数组中。

语法

```
str_split(string,length)
```

参数	描述
string	必需。规定要分割的字符串。
length	可选。规定每个数组元素的长度。默认是 1。

说明

如果 *length* 小于 1，str_split() 函数将返回 false。

如果 *length* 大于字符串的长度，整个字符串将作为数组的唯一元素返回。

例子

例子 1

```
<?php
print_r(str_split("Hello"));
?>
```

输出：

```
Array
(
[0] => H
[1] => e
[2] => l
[3] => l
[4] => o
)
```

例子 2

```
<?php
print_r(str_split("Hello",3));
?>
```

输出：

```
Array
(
[0] => Hel
[1] => lo
)
```

PHP str_word_count() 函数

定义和用法

str_word_count() 函数计算字符串中的单词数。

语法

```
str_word_count(string,return,char)
```

参数	描述
string	必需。规定要检查的字符串。
return	可选。规定 str_word_count() 函数的返回值。可能的值： `0` - 默认。返回找到的单词的数目。 `1` - 返回包含字符串中的单词的数组。 `2` - 返回一个数组，其中的键是单词在字符串中的位置，值是实际的单词。
return	可选。规定被认定为单词的特殊字符。该参数是 PHP 5.1 中新加的。

例子

例子 1

```
<?php
echo str_word_count("Hello world!");
?>
```

输出：

```
2
```

例子 2

```
<?php
print_r(str_word_count("Hello world!",1));
?>
```

输出：

```
Array
(
[0] => Hello
[1] => world
)
```

例子 3

```
<?php
print_r(str_word_count("Hello world!",2));
?>
```

输出：

```
Array
(
[0] => Hello
[6] => world
)
```

例子 4

```
<?php
print_r(str_word_count("Hello world & good morning!",1));
print_r(str_word_count("Hello world & good morning!",1,"&"));
?>
```

输出：

```
Array
(
[0] => Hello
[1] => world
[2] => good
[3] => morning
)

Array
(
[0] => Hello
[1] => world
[2] => &
[3] => good
[4] => morning
)
```

PHP strcasecmp() 函数

定义和用法

strcasecmp() 函数比较两个字符串。

该函数返回：

- 0 - 如果两个字符串相等
- <0 - 如果 string1 小于 string2
- >0 - 如果 string1 大于 string2

语法

```
strcasecmp(string1,string2)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。

提示和注释

注释：该函数是二进制安全的，且对大小写不敏感。

例子

```
<?php
echo strcasecmp("Hello world!","HELLO WORLD!");
?>
```

输出：

```
0
```

PHP strchr() 函数

定义和用法

strchr() 函数搜索一个字符串在另一个字符串中的第一次出现。

该函数返回字符串的其余部分（从匹配点）。如果未找到所搜索的字符串，则返回 false。

该函数是 strstr() 函数的别名。

语法

```
strchr(string,search)
```

参数	描述
string	必需。规定被搜索的字符串。
search	必需。规定所搜索的字符串。如果该参数是数字，则搜索匹配数字 ASCII 值的字符。

提示和注释

注释：该函数是二进制安全的。

注释：该函数对大小写敏感。如需进行大小写不敏感的搜索，请使用 stristr()。

例子

例子 1

```
<?php
echo strchr("Hello world!","world");
?>
```

输出：

```
world!
```

例子 2

在本例中，我们将搜索 "o" 的 ASCII 值所代表的字符：

```
<?php
echo strchr("Hello world!",111);
?>
```

输出：

```
o world!
```

PHP strcmp() 函数

定义和用法

strcmp() 函数比较两个字符串。

该函数返回：

- 0 - 如果两个字符串相等
- <0 - 如果 string1 小于 string2
- >0 - 如果 string1 大于 string2

语法

```
strcmp(string1,string2)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。

提示和注释

注释：该函数是二进制安全的，且对大小写敏感。

例子

```
<?php
echo strcmp("Hello world!","Hello world!");
?>
```

输出：

```
0
```

PHP strcoll() 函数

定义和用法

strcoll() 函数比较两个字符串。

该函数返回：

- 0 - 如果两个字符串相等
- <0 - 如果 string1 小于 string2
- >0 - 如果 string1 大于 string2

字符串的比较会根据本地设置而变化。（A<a 或 A>a）。

语法

```
strcoll(string1,string2)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。

提示和注释

注释：该函数对大小写敏感，但不是二进制安全的。

注释：如果本地设置是 C 或 POSIX，则该函数的工作方式与 strcmp() 相同。

例子

```
<?php
setlocale (LC_COLLATE, 'NL');
echo strcoll("Hello World!","Hello WORLD!");
echo "<br />";

setlocale (LC_COLLATE, 'en_US');
echo strcoll("Hello World!","Hello WORLD!");
?>
```

输出：

```
-1
1
```

PHP strcspn() 函数

定义和用法

strcspn() 函数返回在找到任何指定的字符之前，在字符串查找的字符数。

语法

```
strcspn(string,char,start,length)
```

参数	描述
string	必需。规定要搜索的字符串。
char	必需。规定要查找的字符。
start	可选。规定开始查找的位置。该参数是 PHP 4.3 中新加的。
length	可选。规定字符串的长度（搜索多少字符）。该参数是 PHP 4.3 中新加的。

提示和注释

注释：该函数是二进制安全的。

例子

```
<?php
echo strcspn("Hello world!","w");
?>
```

输出：

```
6
```

PHP strip_tags() 函数

定义和用法

strip_tags() 函数剥去 HTML、XML 以及 PHP 的标签。

语法

```
strip_tags(string,allow)
```

参数	描述
string	必需。规定要检查的字符串。
allow	可选。规定允许的标签。这些标签不会被删除。

提示和注释

注释：该函数始终会剥离 HTML 注释。这点无法通过 *allow* 参数改变。

例子

例子 1

```
<?php
echo strip_tags("Hello <b>world!</b>");
?>
```

输出：

```
Hello world!
```

例子 2

```
<?php
echo strip_tags("Hello <b><i>world!</i></b>","<b>");
?>
```

输出：

```
Hello **world!**
```

PHP stripcslashes() 函数

定义和用法

stripcslashes() 函数删除由 addcslashes() 函数添加的反斜杠。

语法

```
stripcslashes(string)
```

参数	描述
string	必需。规定要检查的字符串。

提示和注释

注释：该函数用于清理从数据库中取回的数据。

例子

```
<?php
echo stripcslashes("Hello, \my na\me is Kai Ji\m.");
?>
```

输出：

```
Hello, my name is Kai Jim.
```

PHP stripslashes() 函数

定义和用法

stripslashes() 函数删除由 addslashes() 函数添加的反斜杠。

语法

```
stripslashes(string)
```

参数	描述
string	必需。规定要检查的字符串。

提示和注释

注释：该函数用于清理从数据库或 HTML 表单中取回的数据。

例子

```
<?php
echo stripslashes("Who\'s John Adams?");
?>
```

输出：

```
Who's John Adams?
```

PHP stripos() 函数

定义和用法

stripos() 函数返回字符串在另一个字符串中第一次出现的位置。

如果没有找到该字符串，则返回 false。

语法

```
stripos(string,find,start)
```

参数	描述
string	必需。规定被搜索的字符串。
find	必需。规定要查找的字符。
start	可选。规定开始搜索的位置。

提示和注释

注释：该函数对大小写不敏感。如需进行对大小写敏感的搜索，请使用 strpos() 函数。

例子

```
<?php
echo stripos("Hello world!","WO");
?>
```

输出：

```
6
```

PHP stristr() 函数

定义和用法

stristr() 函数查找字符串在另一个字符串中第一次出现的位置。

如果成功，则返回字符串的其余部分（从匹配点）。如果没有找到该字符串，则返回 false。

语法

```
stristr(string,search)
```

参数	描述
string	必需。规定被搜索的字符串。
find	必需。规定要查找的字符。如果该参数是数字，则搜索匹配该数字对应的 ASCII 值的字符。

提示和注释

注释：该函数是二进制安全的。

注释：该函数对大小写不敏感。如需对大小写敏感的搜索，请使用 strstr()。

例子

例子 1

```
<?php
echo stristr("Hello world!","WORLD");
?>
```

输出：

```
world!
```

例子 2

```
<?php
echo stristr("Hello world!",111);
?>
```

输出：

```
o world!
```

PHP strlen() 函数

定义和用法

strlen() 函数返回字符串的长度。

语法

```
strlen(string)
```

参数	描述
string	必需。规定要检查的字符串。

例子

```
<?php
echo strlen("Hello world!");
?>
```

输出：

```
12
```

PHP strnatcasecmp() 函数

定义和用法

strnatcasecmp() 函数使用一种"自然"算法来比较两个字符串。

在自然算法中，数字 "2" 小于数字 "10"。在计算机排序中，"2" 大于 "10"，这是因为 "2" 大于 "10" 的第一个数字。

该函数返回：

- 0 - 如果两个字符串相等
- <0 - 如果 string1 小于 string2
- >0 - 如果 string1 大于 string2

语法

```
strnatcasecmp(string1,string2)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。

提示和注释

注释：该函数对大小写不敏感。

例子

```
<?php
echo strnatcasecmp("2Hello world!","10Hello world!");
echo "<br />";
echo strnatcasecmp("10Hello world!","2Hello world!");
?>
```

输出：

```
-1
1
```

PHP strnatcmp() 函数

定义和用法

strnatcmp() 函数使用一种“自然”算法来比较两个字符串。

在自然算法中，数字 "2" 小于数字 "10"。在计算机排序中，"2" 大于 "10"，这是因为 "2" 大于 "10" 的第一个数字。

该函数返回：

- 0 - 如果两个字符串相等
- <0 - 如果 string1 小于 string2
- >0 - 如果 string1 大于 string2

语法

```
strnatcmp(string1,string2)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。

提示和注释

注释：该函数对大小写敏感。

例子

```
<?php
echo strnatcmp("2Hello world!","10Hello world!");
echo "<br />";
echo strnatcmp("10Hello world!","2Hello world!");
?>
```

输出：

```
-1
1
```

PHP strncasecmp() 函数

定义和用法

strncasecmp() 函数比较两个字符串。

该函数返回：

- 0 - 如果两个字符串相等
- <0 - 如果 string1 小于 string2
- >0 - 如果 string1 大于 string2

语法

```
strncasecmp(string1,string2,length)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。
length	必需。规定每个字符串用于比较的字符数。

提示和注释

注释：该函数是二进制安全的，且对大小写不敏感。

例子

```
<?php
echo strncasecmp("Hello world!","Hello earth!",6);
?>
```

输出：

```
0
```

PHP strncmp() 函数

定义和用法

strncmp() 函数比较两个字符串。

该函数返回：

- 0 - 如果两个字符串相等
- <0 - 如果 string1 小于 string2
- >0 - 如果 string1 大于 string2

语法

```
strncmp(string1,string2,length)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。
length	必需。规定每个字符串用于比较的字符数。

提示和注释

注释：该函数是二进制安全的，且对大小写敏感。

例子

```
<?php
echo strncmp("Hello world!","Hello earth!",6);
?>
```

输出：

```
0
```

PHP strpbrk() 函数

定义和用法

strpbrk() 函数在字符串中搜索指定字符中的任意一个。

该函数返回指定字符第一次出现的位置开始的剩余部分。如果没有找到，则返回 false。

语法

```
strpbrk(string,charlist)
```

参数	描述
string	必需。规定被搜索的字符串。
charlist	必需。规定要查找的字符。

提示和注释

注释：该函数对大小写敏感。

例子

```
<?php
echo strpbrk("Hello world!","oe");
?>
```

输出：

```
ello world!
```

PHP strpos() 函数

定义和用法

strpos() 函数返回字符串在另一个字符串中第一次出现的位置。

如果没有找到该字符串，则返回 false。

语法

```
strpos(string,find,start)
```

参数	描述
string	必需。规定被搜索的字符串。
find	必需。规定要查找的字符。
start	可选。规定开始搜索的位置。

提示和注释

注释：该函数对大小写敏感。如需进行对大小写不敏感的搜索，请使用 stripos() 函数。

例子

```
<?php
echo strpos("Hello world!","wo");
?>
```

输出：

```
6
```

PHP strrchr() 函数

定义和用法

strrchr() 函数查找字符串在另一个字符串中最后一次出现的位置，并返回从该位置到字符串结尾的所有字符。

如果成失败，否则返回 false。

语法

```
strrchr(string,char)
```

参数	描述
string	必需。规定被搜索的字符串。
char	必需。规定要查找的字符。如果该参数是数字，则搜索匹配数字 ASCII 值的字符。

提示和注释

注释：该函数是二进制安全的。

例子

例子 1

```
<?php
echo strrchr("Hello world!","world");
?>
```

输出：

```
world!
```

例子 2

```
<?php
echo strrchr("Hello world!",111);
?>
```

输出：

```
orld!
```

PHP strrev() 函数

定义和用法

strrev() 函数反转字符串。

语法

```
strrev(string)
```

参数	描述
string	必需。规定要反转的字符串。

例子

```
<?php
echo strrev("Hello World!");
?>
```

输出：

```
!dlroW olleH
```

PHP strripos() 函数

定义和用法

strripos() 函数查找字符串在另一个字符串中最后一次出现的位置。

如果成功，则返回位置，否则返回 false。

语法

```
strrpos(string,find,start)
```

参数	描述
string	必需。规定被搜索的字符串。
find	必需。规定要查找的字符。
start	可选。规定开始搜索的位置。

提示和注释

注释：该函数对大小写不敏感。如需进行大小写敏感的查找，请使用 strrpos()。

例子

```
<?php
echo strripos("Hello world!","WO");
?>
```

输出：

```
6
```

PHP strrpos() 函数

定义和用法

strrpos() 函数查找字符串在另一个字符串中最后一次出现的位置。

如果成功，则返回位置，否则返回 false。

语法

```
strrpos(string,find,start)
```

参数	描述
string	必需。规定被搜索的字符串。
find	必需。规定要查找的字符。
start	可选。规定开始搜索的位置。

提示和注释

注释：该函数对大小写敏感。如需进行大小写不敏感的查找，请使用 strripos()。

例子

```
<?php
echo strrpos("Hello world!","wo");
?>
```

输出：

```
6
```

PHP strspn() 函数

定义和用法

strspn() 函数返回在字符串中包含的特定字符的数目。

语法

```
strspn(string,charlist,start,length)
```

参数	描述
string	必需。规定被搜索的字符串。
charlist	必需。规定要查找的字符。
start	可选。规定在字符串的何处开始。
length	可选。规定字符串的长度。

提示和注释

注释：该函数是二进制安全的。

例子

例子 1

```
<?php
echo strspn("Hello world!","kHlleo");
?>
```

输出：

```
5
```

例子 2

```
<?php
echo strspn("abcdefand","abc");
?>
```

输出：

```
3
```

PHP strstr() 函数

定义和用法

strstr() 函数搜索一个字符串在另一个字符串中的第一次出现。

该函数返回字符串的其余部分（从匹配点）。如果未找到所搜索的字符串，则返回 false。

语法

```
strstr(string,search)
```

参数	描述
string	必需。规定被搜索的字符串。
search	必需。规定所搜索的字符串。如果该参数是数字，则搜索匹配数字 ASCII 值的字符。

提示和注释

注释：该函数是二进制安全的。

注释：该函数对大小写敏感。如需进行大小写不敏感的搜索，请使用 stristr()。

例子

例子 1

```
<?php
echo strstr("Hello world!","world");
?>
```

输出：

```
world!
```

例子 2

在本例中，我们将搜索 "o" 的 ASCII 值所代表的字符：

```
<?php
echo strstr("Hello world!",111);
?>
```

输出：

```
o world!
```

PHP strtok() 函数

定义和用法

strtok() 函数把字符串分割为更小的字符串。

语法

```
strtok(string,split)
```

参数	描述
string	必需。规定要分割的字符串。
split	必需。规定一个或多个分割字符。

例子

在下面的例子中，请注意，我们仅在第一次调用 strtok() 函数时使用了 *string* 参数。在首次调用后，该函数仅需要 *split* 参数，这是因为它清楚自己在当前函数中所在的位置：

```
<?php
$string = "Hello world. Beautiful day today.";
$token = strtok($string, " ");

while ($token !== false)
  {
  echo "$token<br />";
  $token = strtok(" ");
  }
?>
```

输出：

```
Hello
world.
Beautiful
day
today.
```

PHP strtolower() 函数

定义和用法

strtolower() 函数把字符串转换为小写。

语法

```
strtolower(string)
```

参数	描述
string	必需。规定要转换的字符串。

例子

```
<?php
echo strtolower("Hello WORLD!");
?>
```

输出：

```
hello world!
```

PHP strtoupper() 函数

定义和用法

strtoupper() 函数把字符串转换为大写。

语法

```
strtoupper(string)
```

参数	描述
string	必需。规定要转换的字符串。

例子

```
<?php
echo strtoupper("Hello WORLD!");
?>
```

输出：

```
HELLO WORLD!
```

PHP strtr() 函数

定义和用法

strtr() 函数转换字符串中特定的字符。

语法

```
strtr(string,from,to)
```

或者

```
strtr(string,array)
```

参数	描述
string1	必需。规定要转换的字符串。
from	必需（除非使用数组）。规定要改变的字符。
to	必需（除非使用数组）。规定要改变为的字符。
array	必需（除非使用 *from* 和 *to*）。一个数组，其中的键是原始字符，值是目标字符。

说明

如果 *from* 和 *to* 的长度不同，则格式化为最短的长度。

例子

例子 1

```
<?php
echo strtr("Hilla Warld","ia","eo");
?>
```

输出：

```
Hello World
```

例子 2

```
<?php
$arr = array("Hello" => "Hi", "world" => "earth");
echo strtr("Hello world",$arr);
?>
```

输出：

```
Hi earth
```

PHP substr() 函数

定义和用法

substr() 函数返回字符串的一部分。

语法

```
substr(_string_,_start_,_length_)
```

参数	描述
string	必需。规定要返回其中一部分的字符串。
start	必需。规定在字符串的何处开始。 `正数` - 在字符串的指定位置开始 `负数` - 在从字符串结尾的指定位置开始 `0` - 在字符串中的第一个字符处开始
length	可选。规定要返回的字符串长度。默认是直到字符串的结尾。 `正数` - 从 start 参数所在的位置返回 `负数` - 从字符串末端返回

提示和注释

注释：如果 *start* 是负数且 *length* 小于等于 *start*，则 *length* 为 0。

例子

例子 1

```
<?php
echo substr("Hello world!",6);
?>
```

输出：

```
world!
```

例子 2

```
<?php
echo substr("Hello world!",6,5);
?>
```

输出：

```
world
```

PHP substr_compare() 函数

定义和用法

substr_compare() 函数从指定的开始长度比较两个字符串。

该函数返回：

- 0 - 如果两字符串相等
- <0 - 如果 string1 （从开始位置）小于 string2
- >0 - 如果 string1 （从开始位置）大于 string2

语法

```
substr_compare(string1,string2,startpos,length,case)
```

参数	描述
string1	必需。规定要比较的第一个字符串。
string2	必需。规定要比较的第二个字符串。
startpos	可选。规定在 *string1* 中的何处开始比较。
length	可选。规定在 *string1* 中参与比较的字符数。
case	可选。规定是否指定大小写比较。默认是 FALSE （对大小写敏感）。

说明

如果 *length* 大于或等于 *string1* 的长度，则函数返回 false。

例子

例子 1

```
<?php
echo substr_compare("Hello world","Hello world",0);
?>
```

输出：

```
0
```

例子 2

```
<?php
echo substr_compare("Hello world","world",6);
?>
```

输出：

```
0
```

例子 3

```
<?php
echo substr_compare("Hello world","WORLD",6,TRUE);
?>
```

输出：

```
0
```

PHP substr_count() 函数

定义和用法

substr_count() 函数计算子串在字符串中出现的次数。

语法

```
substr_count(_string_,_substring_,_start_,_length_)
```

参数	描述
string	必需。规定要检查的字符串。
substring	必需。规定要检索的字符串。
start	可选。规定在字符串中何处开始搜索。
length	可选。规定搜索的长度。

例子

```
<?php
echo substr_count("Hello world. The world is nice","world");
?>
```

输出：

```
2
```

PHP substr_replace() 函数

定义和用法

substr_replace() 函数把字符串的一部分替换为另一个字符串。

语法

```
substr_replace(string,replacement,start,length)
```

参数	描述
string	必需。规定要检查的字符串。
replacement	必需。规定要插入的字符串。
start	必需。规定在字符串的何处开始替换。 `正数` - 在第 *start* 个偏移量开始替换 `负数` - 在从字符串结尾的第 *start* 个偏移量开始替换 `0` - 在字符串中的第一个字符处开始替换
charlist	可选。规定要替换多少个字符。 `正数` - 被替换的字符串长度 `负数` - 从字符串末端开始的被替换字符数 `0` - 插入而非替换

提示和注释

注释：如果 *start* 是负数且 *length* 小于等于 *start*，则 *length* 为 0。

例子

```
<?php
echo substr_replace("Hello world","earth",6);
?>
```

输出：

```
Hello earth
```

PHP trim() 函数

定义和用法

trim() 函数从字符串的两端删除空白字符和其他预定义字符。

语法

```
trim(string,charlist)
```

参数	描述
string	必需。规定要检查的字符串。
charlist	可选。规定要转换的字符串。如果省略该参数，则删除以下所有字符： " \0 " - NULL " \t " - tab " \n " - new line " \x0B " - 纵向列表符 " \r " - 回车 " " - 普通空白字符

例子

例子 1

```
<html>
<body>
<?php
$str = "    Hello World!    ";
echo "Without trim: " . $str;
echo "<br />";
echo "With trim: " . trim($str);
?>
<body>
<html>
```

输出：

```
Without trim: Hello World!
With trim: Hello World!
```

HTML 源码：

```
<html>
<body>
Without trim:     Hello World!    <br />With trim: Hello World!
</body>
</html>
```

例子 2

```
<?php
$str = "\r\nHello World!\r\n";
echo "Without trim: " . $str;
echo "<br />";
echo "With trim: " . trim($str);
?>
```

输出：

```
Without trim: Hello World!
With trim: Hello World!
```

HTML 源码：

```
<html>
<body>
Without trim:
Hello World!
<br />With trim: Hello World!
</body>
</html>
```

PHP ucfirst() 函数

定义和用法

ucfirst() 函数把字符串中的首字符转换为大写。

语法

```
ucfirst(string)
```

参数	描述
string	必需。规定要转换的字符串。

例子

```
<?php
echo ucfirst("hello world");
?>
```

输出：

```
Hello world
```

PHP ucwords() 函数

定义和用法

ucwords() 函数把字符串中每个单词的首字符转换为大写。

语法

```
ucwords(string)
```

参数	描述
string	必需。规定要转换的字符串。

例子

```
<?php
echo ucwords("hello world");
?>
```

输出：

```
Hello World
```

PHP vfprintf() 函数

定义和用法

vfprintf() 函数把格式化的字符串写到指定的输出流。

与 fprintf() 不同，vfprintf() 中的 *arg* 参数位于数组中。数组的元素会被插入主字符串的百分比 (%) 符号处。该函数是逐步执行的。在第一个 % 符号中，插入 arg1，在第二个 % 符号处，插入 arg2，依此类推。

该函数返回被写的字符串的长度。

语法

```
vfprintf(stream,format,argarray)
```

参数	描述
stream	必需。规定在何处写/输出字符串。
format	必需。转换格式。
argarray	必需。带有参数的一个数组，这些参数会被插到 format 字符串中的 % 符号处。

说明

参数 *format* 是转换的格式，以百分比符号 ("%") 开始到转换字符结束。下面的可能的 *format* 值：

- %% - 返回百分比符号
- %b - 二进制数
- %c - 依照 ASCII 值的字符
- %d - 带符号十进制数
- %e - 可续计数法（比如 1.5e+3）
- %u - 无符号十进制数
- %f - 浮点数(local settings aware)
- %F - 浮点数(not local settings aware)
- %o - 八进制数
- %s - 字符串
- %x - 十六进制数（小写字母）

- %X - 十六进制数（大写字母）

提示和注释

注释：如果 % 符号多于 *arg* 参数，则您必须使用占位符。占位符插到 % 符号后面，由数字和 "$" 组成。请参见例子 3。

提示： 相关函数： fprintf()、 printf()、 sprintf()、 vprintf() 以及 vsprintf()。

例子

例子 1

```
<?php
$str = "Hello";
$number = 123;
$txt = sprintf("%s world. Day number %u",$str,$number);
echo $txt;
?>
```

输出：

```
Hello world. Day number 123
```

例子 2

```
<?php
$number = 123;
$txt = sprintf("%f",$number);
echo $txt;
?>
```

输出：

```
123.000000
```

例子 3

```
<?php
$number = 123;
$txt = sprintf("With 2 decimals: %1\$.2f<br />With no decimals: %1\$u",$number);
echo $txt;
?>
```

输出：

```
With 2 decimals: 123.00
With no decimals: 123
```

PHP vprintf() 函数

定义和用法

vprintf() 函数输出格式化的字符串。

与 printf() 不同，vprintf() 中的 *arg* 参数位于数组中。数组的元素会被插入主字符串的百分比 (%) 符号处。该函数是逐步执行的。在第一个 % 符号中，插入 arg1，在第二个 % 符号处，插入 arg2，依此类推。

语法

```
vprintf(format,argarray)
```

参数	描述
format	必需。转换格式。
argarray	必需。带有参数的一个数组，这些参数会被插到 format 字符串中的 % 符号处。

说明

参数 *format* 是转换的格式，以百分比符号 ("%") 开始到转换字符结束。下面的可能的 *format* 值：

- %% - 返回百分比符号
- %b - 二进制数
- %c - 依照 ASCII 值的字符
- %d - 带符号十进制数
- %e - 可续计数法（比如 1.5e+3）
- %u - 无符号十进制数
- %f - 浮点数(local settings aware)
- %F - 浮点数(not local settings aware)
- %o - 八进制数
- %s - 字符串
- %x - 十六进制数（小写字母）
- %X - 十六进制数（大写字母）

提示和注释

注释：如果 % 符号多于 *arg* 参数，则您必须使用占位符。占位符插到 % 符号后面，由数字和 "$" 组成。请参见例子 3。

提示： 相关函数： fprintf()、 printf()、 sprintf()、 vfprintf() 以及 vsprintf()。

例子

例子 1

```
<?php
$str = "Hello";
$number = 123;
vprintf("%s world. Day number %u",array($str,$number));
?>
```

输出：

```
Hello world. Day number 123
```

例子 2

```
<?php
$num1 = 123;
$num2 = 456;
vprintf("%f%f",array($num1,$num2));
?>
```

输出：

```
123.000000456.000000
```

例子 3

使用占位符：

```
<?php
$number = 123;
vprintf("With 2 decimals: %1\$.2f<br />With no decimals: %1\$u",array($number));
?>
```

输出：

```
With 2 decimals: 123.00
With no decimals: 123
```

PHP vsprintf() 函数

定义和用法

vsprintf() 函数把格式化字符串写入变量中。

与 sprintf() 不同，vsprintf() 中的 *arg* 参数位于数组中。数组的元素会被插入主字符串的百分比 (%) 符号处。该函数是逐步执行的。在第一个 % 符号中，插入 arg1，在第二个 % 符号处，插入 arg2，依此类推。

语法

```
vsprintf(format,argarray)
```

参数	描述
format	必需。转换格式。
argarray	必需。带有参数的一个数组，这些参数会被插到 format 字符串中的 % 符号处。

说明

参数 *format* 是转换的格式，以百分比符号 ("%") 开始到转换字符结束。下面的可能的 *format* 值：

- %% - 返回百分比符号
- %b - 二进制数
- %c - 依照 ASCII 值的字符
- %d - 带符号十进制数
- %e - 可续计数法（比如 1.5e+3）
- %u - 无符号十进制数
- %f - 浮点数(local settings aware)
- %F - 浮点数(not local settings aware)
- %o - 八进制数
- %s - 字符串
- %x - 十六进制数（小写字母）
- %X - 十六进制数（大写字母）

提示和注释

注释：如果 % 符号多于 *arg* 参数，则您必须使用占位符。占位符插到 % 符号后面，由数字和 "$" 组成。请参见例子 3。

提示： 相关函数： fprintf()、 printf()、 sprintf()、 vfprintf() 以及 vprintf()。

例子

例子 1

```
<?php
$str = "Hello";
$number = 123;
$txt = vsprintf("%s world. Day number %u",array($str,$number));
echo $txt;
?>
```

输出：

```
Hello world. Day number 123
```

例子 2

```
<?php
$num1 = 123;
$num2 = 456;
$txt = vsprintf("%f%f",array($num1,$num2));
echo $txt;
?>
```

输出：

```
123.000000456.000000
```

例子 3

使用占位符：

```
<?php
$number = 123;
$txt = vsprintf("With 2 decimals: %1\$.2f
<br />With no decimals: %1\$u",array($number));
echo $txt;
?>
```

输出：

```
With 2 decimals: 123.00
With no decimals: 123
```

PHP wordwrap() 函数

定义和用法

wordwrap() 函数按照指定长度对字符串进行折行处理。

如果成功，则返回折行后的字符串。如果失败，则返回 false。

语法

```
wordwrap(string,width,break,cut)
```

参数	描述
string	必需。规定要进行折行的字符串。
width	可选。规定最大行宽度。默认是 75。
break	可选。规定作为分隔符使用的字符（字串断开字符）。默认是 "\n"。
cut	可选。规定是否对大约指定宽度的单词进行折行。默认是 FALSE (no-wrap)。

例子

例子 1

```
<?php
$str = "An example on a long word is: Supercalifragulistic";
echo wordwrap($str,15);
?>
```

浏览器输出：

```
An example on a long word is: Supercalifragulistic
```

HTML 源代码：

```
<html>
<body>
An example on a
long word is:
Supercalifragulistic
</body>
</html>
```

例子 2

```
<?php
$str = "An example on a long word is: Supercalifragulistic";
echo wordwrap($str,15,"<br />\n");
?>
```

输出：

```
An example on a
long word is:
Supercalifragulistic
```

例子 3

```
<?php
$str = "An example on a long word is: Supercalifragulistic";
echo wordwrap($str,15,"<br />\n",TRUE);
?>
```

输出：

```
An example on a
long word is:
Supercalifragul
istic
```

PHP XML Parser 函数

PHP XML Parser 简介

XML 函数允许我们解析 XML 文档，但无法对其进行验证。

XML 是一种用于标准结构化文档交换的数据格式。您可以在我们的 XML 教程 中找到更多有关 XML 的信息。

该扩展使用 Expat XML 解析器。

Expat 是一种基于事件的解析器，它把 XML 文档视为一系列事件。当某个事件发生时，它调用一个指定的函数处理它。

Expat 是无验证的解析器，忽略任何链接到文档的 DTD。但是，如果文档的形式不好，则会以一个错误消息结束。

由于它基于事件，且无验证，Expat 具有快速并适合 web 应用程序的特性。

XML 解析器函数允许我们创建 XML 解析器，并为 XML 事件定义句柄。

安装

XML 函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

PHP XML Parser 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
utf8_decode()	把 UTF-8 字符串解码为 ISO-8859-1。	3
utf8_encode()	把 ISO-8859-1 字符串编码为 UTF-8。	3
xml_error_string()	获取 XML 解析器的错误描述。	3
xml_get_current_byte_index()	获取 XML 解析器的当前字节索引。	3
xml_get_current_column_number()	获取 XML 解析器的当前列号。	3
xml_get_current_line_number()	获取 XML 解析器的当前行号。	3
xml_get_error_code()	获取 XML 解析器错误代码。	3
xml_parse()	解析 XML 文档。	3
xml_parse_into_struct()	把 XML 数据解析到数组中。	3
xml_parser_create_ns()	创建带有命名空间支持的 XML 解析器。	4
xml_parser_create()	创建 XML 解析器。	3
xml_parser_free()	释放 XML 解析器。	3
xml_parser_get_option()	从 XML 解析器获取选项设置信息。	3
xml_parser_set_option()	为 XML 解析进行选项设置。	3
xml_set_character_data_handler()	建立字符数据处理器。	3
xml_set_default_handler()	建立默认的数据处理器。	3
xml_set_element_handler()	建立起始和终止元素处理器。	3
xml_set_end_namespace_decl_handler()	建立终止命名空间声明处理器。	4
xml_set_external_entity_ref_handler()	建立外部实体处理器。	3
xml_set_notation_decl_handler()	建立注释声明处理器。	3
xml_set_object()	在对象中使用 XML 解析器。	4
xml_set_processing_instruction_handler()	建立处理指令（PI）处理器。	3
xml_set_start_namespace_decl_handler()	建立起始命名空间声明处理器。	4
xml_set_unparsed_entity_decl_handler()	建立未解析实体定义声明处理器。	3

PHP XML Parser 常量

Constant
XML_ERROR_NONE (integer)
XML_ERROR_NO_MEMORY (integer)
XML_ERROR_SYNTAX (integer)
XML_ERROR_NO_ELEMENTS (integer)
XML_ERROR_INVALID_TOKEN (integer)
XML_ERROR_UNCLOSED_TOKEN (integer)
XML_ERROR_PARTIAL_CHAR (integer)
XML_ERROR_TAG_MISMATCH (integer)
XML_ERROR_DUPLICATE_ATTRIBUTE (integer)
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (integer)
XML_ERROR_PARAM_ENTITY_REF (integer)
XML_ERROR_UNDEFINED_ENTITY (integer)
XML_ERROR_RECURSIVE_ENTITY_REF (integer)
XML_ERROR_ASYNC_ENTITY (integer)
XML_ERROR_BAD_CHAR_REF (integer)
XML_ERROR_BINARY_ENTITY_REF (integer)
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (integer)
XML_ERROR_MISPLACED_XML_PI (integer)
XML_ERROR_UNKNOWN_ENCODING (integer)
XML_ERROR_INCORRECT_ENCODING (integer)
XML_ERROR_UNCLOSED_CDATA_SECTION (integer)
XML_ERROR_EXTERNAL_ENTITY_HANDLING (integer)
XML_OPTION_CASE_FOLDING (integer)
XML_OPTION_TARGET_ENCODING (integer)
XML_OPTION_SKIP_TAGSTART (integer)
XML_OPTION_SKIP_WHITE (integer)

PHP utf8_decode() 函数

定义和用法

utf8_decode() 函数把 UTF-8 字符串解码为 ISO-8859-1。

该函数把用 UTF-8 方式编码的 ISO-8859-1 字符串转换成单字节的 ISO-8859-1 字符串。

如果成功，该函数将返回解码字符串；否则返回 false。

语法

```
utf8_decode(string)
```

参数	描述
string	必需。规定要解码的字符串。

PHP utf8_encode() 函数

定义和用法

utf8_encode() 函数把 ISO-8859-1 字符串编码为 UTF-8。

Unicode 是全球标准，已经发展到能够通过唯一的编码来描述所有语言中的字符，外加大量的符号。

不过，并不是总能可靠地在计算机之间传递 Unicode 字符。UTF-8 可用于在计算机之间传输 Unicode 字符。

如果成功，该函数将返回编码字符串；否则返回 false。

语法

```
utf8_encode(string)
```

参数	描述
string	必需。规定要编码的字符串。

PHP xml_error_string() 函数

定义和用法

xml_error_string() 函数获取 XML 解析器的错误描述。

语法

```
xml_error_string(errorcode)
```

参数	描述
errorcode	必需。规定要使用的错误代码。该错误码是 xml_get_error_code() 函数的返回值。

说明

返回与 *errorcode* 描述的错误代码参数对应的文本描述字符串，若没有与之对应的描述，则返回 false。

例子

```
<?php
//无效的 xml 文件
$xmlfile = 'test.xml';

$xmlparser = xml_parser_create();

// open a file and read data
$fp = fopen($xmlfile, 'r');
while ($xmldata = fread($fp, 4096))
  {
  // parse the data chunk
  if (!xml_parse($xmlparser,$xmldata,feof($fp)))
    {
    die( print "ERROR: "
    . xml_error_string(xml_get_error_code($xmlparser))
    . "<br />"
    . "Line: "
    . xml_get_current_line_number($xmlparser)
    . "<br />"
    . "Column: "
    . xml_get_current_column_number($xmlparser)
    . "<br />");
    }
  }

xml_parser_free($xmlparser);
?>
```

输出：

```
ERROR: Mismatched tag
Line: 8
Column: 51
```

PHP xml_get_current_byte_index() 函数

定义和用法

xml_get_current_byte_index() 函数获取 XML 解析器的当前字节索引。

语法

```
xml_get_current_byte_index(parser)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。

说明

如果 *parser* 没有指向一个合法的解析器，该函数将返回 false，否则将返回解析器当前在其数据缓冲区中的字节索引（起始值为 0）。

例子

```
<?php
//无效的 xml 文件
$xmlfile = 'test.xml';

$xmlparser = xml_parser_create();

// 打开文件并读取数据
$fp = fopen($xmlfile, 'r');
while ($xmldata = fread($fp, 4096))
  {
  // parse the data chunk
  if (!xml_parse($xmlparser,$xmldata,feof($fp)))
    {
    die( print "ERROR: "
    . xml_error_string(xml_get_error_code($xmlparser))
    . "<br />"
    . "Line: "
    . xml_get_current_line_number($xmlparser)
    . "<br />"
    . "Column: "
    . xml_get_current_column_number($xmlparser)
    . "<br />"
    . "Byte Index: "
    . xml_get_current_byte_index($xmlparser)
    . "<br />");
    }
  }

xml_parser_free($xmlparser);
?>
```

输出：

```
ERROR: Mismatched tag
Line: 8
Column: 51
Byte Index: 96
```

PHP xml_get_current_line_number() 函数

定义和用法

xml_get_current_line_number() 函数获取 XML 解析器的当前行号。

语法

```
xml_get_current_line_number(parser)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。

说明

如果 *parser* 参数没有指向一个合法的解析器，该函数将返回 FALSE，否则将返回指定解析器在其缓存中的当前行号。

例子

```
<?php
//无效 xml 文件
$xmlfile = 'test.xml';

$xmlparser = xml_parser_create();

// 打开文件并读取数据
$fp = fopen($xmlfile, 'r');
while ($xmldata = fread($fp, 4096))
  {
  // parse the data chunk
  if (!xml_parse($xmlparser,$xmldata,feof($fp)))
    {
    die( print "ERROR: "
    . xml_error_string(xml_get_error_code($xmlparser))
    . "<br />"
    . "Line: "
    . xml_get_current_line_number($xmlparser)
    . "<br />"
    . "Column: "
    . xml_get_current_column_number($xmlparser)
    . "<br />");
    }
  }

xml_parser_free($xmlparser);
?>
```

输出：

```
ERROR: Mismatched tag
Line: 8
Column: 61
```

PHP xml_get_error_code() 函数

定义和用法

xml_get_error_code() 函数获取 XML 解析器错误代码。

如果成功，则返回错误代码。否则，返回 false。

语法

```
xml_get_error_code(parser)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。

例子

```
<?php
//无效的 xml 文件
$xmlfile = 'test.xml';

$xmlparser = xml_parser_create();

// 打开文件并读取数据
$fp = fopen($xmlfile, 'r');
while ($xmldata = fread($fp, 4096))
  {
  // parse the data chunk
  if (!xml_parse($xmlparser,$xmldata,feof($fp)))
    {
    die( print "ERROR: "
    . xml_get_error_code($xmlparser)
    . "<br />"
    . "Line: "
    . xml_get_current_line_number($xmlparser)
    . "<br />"
    . "Column: "
    . xml_get_current_column_number($xmlparser)
    . "<br />");
    }
  }

xml_parser_free($xmlparser);
?>
```

输出：

```
ERROR: 76
Line: 8
Column: 61
```

PHP xml_parse() 函数

定义和用法

xml_parse() 函数解析 XML 文档。

如果成功，则返回 true。否则，返回 false。

语法

```
xml_parse(parser,xml,end)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
xml	必需。规定要解析的 XML 数据。
end	可选。如果该参数是 true，则 xml 参数中的数据为当前解析中最后一段数据。

提示和注释

提示：要创建 XML 解析器，请使用 xml_parser_create() 函数。

例子

```
<?php
//无效的 xml 文件
$xmlfile = 'test.xml';

$xmlparser = xml_parser_create();

// 打开文件并读取数据
$fp = fopen($xmlfile, 'r');
while ($xmldata = fread($fp, 4096))
  {
  // parse the data chunk
  if (!xml_parse($xmlparser,$xmldata,feof($fp)))
    {
    die( print "ERROR: "
    . xml_get_error_code($xmlparser)
    . "<br />"
    . "Line: "
    . xml_get_current_line_number($xmlparser)
    . "<br />"
    . "Column: "
    . xml_get_current_column_number($xmlparser)
    . "<br />");
    }
  }

xml_parser_free($xmlparser);
?>
```

输出：

```
ERROR: 76
Line: 8
Column: 61
```

PHP xml_parse_into_struct() 函数

定义和用法

xml_parse_into_struct() 函数把 XML 数据解析到数组中。

该函数把 XML 数据解析到 2 个数组中：

- Value 数组 - 包含来自被解析的 XML 的数据
- Index 数组 - 包含指向Value 数组中值的位置的指针

如果成功，则该函数返回 1。否则返回 0。

语法

```
xml_parse_into_struct(parser,xml,value_arr,index_arr)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
xml	必需。规定要解析的 XML 数据。
value_arr	必需。规定 XML 数据的目标数组。
index_arr	可选。规定 index 数据的目标数组。

提示和注释

注释：xml_parse_into_struct() 若失败返回 0，成功则返回 1。这和 false 与 true 不同，使用例如 === 运算符时要注意。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php
//无效 xml 文件
$xmlfile = 'test.xml';
$xmlparser = xml_parser_create();

// 打开文件并读取数据
$fp = fopen($xmlfile, 'r');
$xmldata = fread($fp, 4096);

xml_parse_into_struct($xmlparser,$xmldata,$values);

xml_parser_free($xmlparser);
print_r($values);
?>
```

输出：

```
Array
(
[0] => Array
  (
  [tag] => NOTE
  [type] => open
  [level] => 1
  [value] =>
  )
  [1] => Array
    (
    [tag] => TO
    [type] => complete
    [level] => 2
    [value] => George
    )
[2] => Array
  (
  [tag] => NOTE
  [value] =>
  [type] => cdata
  [level] => 1
  )
  [3] => Array
    (
    [tag] => FROM
    [type] => complete
    [level] => 2
    [value] => John
    )
[4] => Array
  (
  [tag] => NOTE
  [value] =>
  [type] => cdata
  [level] => 1
  )
  [5] => Array
    (
    [tag] => HEADING
    [type] => complete
    [level] => 2
    [value] => Reminder
    )
[6] => Array
  (
  [tag] => NOTE
  [value] =>
```

```
    [type] => cdata
    [level] => 1
    )
    [7] => Array
      (
      [tag] => BODY
      [type] => complete
      [level] => 2
      [value] => Don't forget the meeting!
      )
  [8] => Array
    (
    [tag] => NOTE
    [value] =>
    [type] => cdata
    [level] => 1
    )
  [9] => Array
    (
    [tag] => NOTE
    [type] => close
    [level] => 1
    )
  )
```

PHP xml_parser_create_ns() 函数

定义和用法

xml_parser_create_ns() 函数创建带有命名空间支持的 XML 解析器。

该函数建立一个新的 XML 解析器并返回可被其它 XML 函数使用的资源句柄。

语法

```
xml_parser_create_ns(encoding,separator)
```

参数	描述
encoding	可选。规定输出编码。
encoding	可选。规定标签名和命名空间的输出分隔符。默认是 ":"。

说明

可选参数 *encoding* 在 PHP 4 中用来指定要被解析的 XML 输入的字符编码方式。

PHP 5 开始，自动侦测输入的 XML 的编码，因此 *encoding* 参数仅用来指定解析后输出数据的编码。

在 PHP 4 中，默认输出的编码与输入数据的编码是相同的。如果传递了空字符串，解析器会尝试搜索头 3 或 4 个字节以确定文档的编码。

在 PHP 5.0.0 和 5.0.1 中，默认输出的字符编码是 ISO-8859-1，而 PHP 5.0.2 及以上版本是 UTF-8。

解析器支持的编码有 ISO-8859-1, UTF-8 和 US-ASCII。

提示和注释

提示：要释放 XML 解析器，请使用 xml_parser_free() 函数。

提示：要创建没有命名空间支持的 XML 解析器，请使用 xml_parser_create() 函数。

例子

```
<?php
$xmlparser = xml_parser_create_ns();

xml_parser_free($xmlparser);
?>
```

PHP xml_parser_create() 函数

定义和用法

xml_parser_create() 函数创建 XML 解析器。

该函数建立一个新的 XML 解析器并返回可被其它 XML 函数使用的资源句柄。

语法

```
xml_parser_create(encoding)
```

参数	描述
encoding	可选。规定输出编码。

说明

可选参数 *encoding* 在 PHP 4 中用来指定要被解析的 XML 输入的字符编码方式。

PHP 5 开始，自动侦测输入的 XML 的编码，因此 *encoding* 参数仅用来指定解析后输出数据的编码。

在 PHP 4 中，默认输出的编码与输入数据的编码是相同的。如果传递了空字符串，解析器会尝试搜索头 3 或 4 个字节以确定文档的编码。

在 PHP 5.0.0 和 5.0.1 中，默认输出的字符编码是 ISO-8859-1，而 PHP 5.0.2 及以上版本是 UTF-8。

解析器支持的编码有 ISO-8859-1, UTF-8 和 US-ASCII。

提示和注释

提示：要释放 XML 解析器，请使用 xml_parser_free() 函数。

提示：要创建带有命名空间支持的 XML 解析器，请使用 xml_parser_create_ns() 函数。

例子

```
<?php
$xmlparser = xml_parser_create();

xml_parser_free($xmlparser);
?>
```

PHP xml_parser_free() 函数

定义和用法

xml_parser_free() 函数释放 XML 解析器。

如果成功，则返回 true。否则返回 false。

语法

```
xml_parser_free(parser)
```

参数	描述
parser	必需。规定要释放的 XML 解析器。

提示和注释

提示：要创建 XML 解析器，请使用 xml_parser_create() 函数。

例子

```
<?php
$xmlparser = xml_parser_create();

xml_parser_free($xmlparser);
?>
```

PHP xml_parser_get_option() 函数

定义和用法

xml_parser_get_option() 函数从 XML 解析器获取选项设置信息。

语法

```
xml_parser_get_option(parser,option)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
option	必需。规定要获取的设置选项名称。可能的值： `XML_OPTION_CASE_FOLDING` `XML_OPTION_TARGET_ENCODING`

说明

如果 *parser* 参数没有指向一个合法的解析器或者 *option* 参数无效，该函数将返回 FALSE（同时产生 E_WARNING 警告）。否则将返回指定设置选项的值。

例子

```
<?php
$xmlparser = xml_parser_create();

echo xml_parser_get_option($xmlparser, XML_OPTION_CASE_FOLDING);

xml_parser_free($xmlparser);
?>
```

PHP xml_parser_set_option() 函数

定义和用法

xml_parser_set_option() 函数为 XML 解析器进行选项设置。

如果成功，则返回 true。如果失败，则返回 false。

语法

```
xml_parser_set_option(parser,option,value)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
option	必需。规定要设置的设置选项名称。可能的值： XML_OPTION_CASE_FOLDING XML_OPTION_SKIP_TAGSTART XML_OPTION_SKIP_WHITE XML_OPTION_TARGET_ENCODING
value	必需。规定选项的新值。

例子

```
<?php
$xmlparser = xml_parser_create();

xml_parser_set_option($xmlparser, XML_OPTION_SKIP_WHITE, 1);

xml_parser_free($xmlparser);
?>
```

PHP xml_set_character_data_handler() 函数

定义和用法

xml_set_character_data_handler() 函数建立字符数据处理器。

该函数规定当解析器在 XML 文件中找到字符数据时所调用的函数。

如果处理器被成功的建立，该函数将返回 true；否则返回 false。

语法

```
xml_set_character_data_handler(parser,handler)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
handler	必需。规定作为事件处理器使用的函数。

由 *handler* 参数规定的函数必须有两个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
data	必需。规定包含字符数据的变量。

说明

handler 参数也可以是一个数组，其中包含对象引用和方法名。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php

$parser=xml_parser_create();

function char($parser,$data)
  {
  echo $data;
  }

xml_set_character_data_handler($parser,"char");
$fp=fopen("test.xml","r");

while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

xml_parser_free($parser);
?>
```

输出：

```
George John Reminder Don't forget the meeting!
```

PHP xml_set_default_handler() 函数

定义和用法

xml_set_default_handler() 函数为 XML 解析器建立默认的数据处理器。

该函数规定在只要解析器在 XML 文件中找到数据时，所调用的函数。

如果处理器被成功的建立，该函数将返回 true；否则返回 false。

语法

```
xml_set_default_handler(parser,handler)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
handler	必需。规定作为事件处理器使用的函数。

由 *handler* 参数规定的函数必须有三个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
data	必需。规定包含数据的变量。

说明

handler 参数也可以是一个数组，其中包含对象引用和方法名。

例子

XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP 代码：

```
<?php

$parser=xml_parser_create();

function default($parser,$data)
  {
  echo $data;
  }

xml_set_default_handler($parser,"default");
$fp=fopen("test.xml","r");

while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

xml_parser_free($parser);
?>
```

输出：

```
George John Reminder Don't forget the meeting!
```

如果在浏览器中查看源代码，会看到下列 HTML：

```
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

PHP xml_set_element_handler() 函数

定义和用法

xml_set_element_handler() 函数建立起始和终止元素处理器。

如果处理器被成功的建立，该函数将返回 true；否则返回 false。

语法

```
xml_set_element_handler(parser,start,end)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
start	必需。规定在元素开始调用的函数。
end	必需。规定在元素结束调用的函数。

由 *start* 参数规定的函数必须有三个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
name	必需。规定一个变量，包含元素的名称，这个元素触发该函数。
data	必需。规定一个数组，包含元素属性。

由 *end* 参数规定的函数必须有三个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
name	必需。规定一个变量，包含元素的名称，这个元素触发该函数。

说明

start 和 *end* 参数也可以是一个数组，其中包含对象引用和方法名。

例子

```
<?php

$parser=xml_parser_create();

function start($parser,$element_name,$element_attrs)
  {
  switch($element_name)
    {
    case "NOTE":
    echo "-- Note --<br />";
    break;
    case "TO":
    echo "To: ";
    break;
    case "FROM":
    echo "From: ";
    break;
    case "HEADING":
    echo "Heading: ";
    break;
    case "BODY":
    echo "Message: ";
    }
  }

function stop($parser,$element_name)
  {
  echo "<br />";
  }

function char($parser,$data)
  {
  echo $data;
  }

xml_set_element_handler($parser,"start","stop");
xml_set_character_data_handler($parser,"char");
$fp=fopen("test.xml","r");

while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

xml_parser_free($parser);

?>
```

输出：

```
-- Note --
To: George
From: John
Heading: Reminder
Message: Don't forget the meeting!
```

PHP xml_set_external_entity_ref_handler() 函数

定义和用法

xml_set_external_entity_ref_handler() 函数规定当解析器在 XML 文档中找到外部实体时被调用的函数。

如果处理器被成功的建立，该函数将返回 true；否则返回 false。

语法

```
xml_set_external_entity_ref_handler(parser,handler)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
handler	必需。规定当解析器找到外部实体时被调用用的函数。

由 *handler* 参数规定的函数必须有六个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
name	必需。规定包含外部实体名称的变量。
base	必需。规定一个变量，包含解析外部实体的系统标识符（system_id ）的基础。当前该参数通常都被设置为空字符串。
system_id	必需。规定包含外部实体的系统标识符的变量。
public_id	必需。规定包含外部实体的公共标识符的变量。

说明

handler 参数也可以是一个数组，其中包含对象引用和方法名。

例子

```
<?php

$parser=xml_parser_create();

function char($parser,$data)
  {
  echo $data;
  }

function ext_ent_handler($parser,$ent,$base,$sysID,$pubID)
  {
  echo "$ent";
  echo "$sysID";
  echo "$pubID";
  }

xml_set_character_data_handler($parser,"char");
xml_set_external_entity_ref_handler($parser, "ext_ent_handler");
$fp=fopen("test.xml","r");

while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

xml_parser_free($parser);

?>
```

PHP xml_set_notation_decl_handler() 函数

定义和用法

xml_set_notation_decl_handler() 函数规定当解析器在 XML 文档中找到符号声明时被调用的函数。

如果处理器被成功的建立，该函数将返回 true；否则返回 false。

注释："符号声明"，英文为 notation declaration，也有部分文献译为"注释声明"。

语法

```
xml_set_notation_decl_handler(parser,handler)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
handler	必需。规定当解析器找到符号声明时被调用的函数。

由 *handler* 参数规定的函数必须有六个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
name	必需。规定包含实体名称的变量。
base	必需。规定一个变量，包含解析实体的系统标识符（system_id ）的基础。当前该参数通常都被设置为空字符串。
system_id	必需。规定包含实体的系统标识符的变量。
public_id	必需。规定包含实体的公共标识符的变量。
notation	必需。规定一个变量，包含标识实体数据类型的符号。

说明

handler 参数也可以是一个数组，其中包含对象引用和方法名。

例子

```
<?php

$parser=xml_parser_create();

function char($parser,$data)
  {
  echo $data;
  }

function not_decl_handler($parser,$not,$base,$sysID,$pubID)
  {
  echo "$not<br />";
  echo "$sysID<br />";
  echo "$pubID<BR />";
  }

xml_set_character_data_handler($parser,"char");
xml_set_notation_decl_handler($parser, "not_decl_handler");
$fp=fopen("test.xml","r");

while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

xml_parser_free($parser);

?>
```

PHP xml_set_object() 函数

定义和用法

xml_set_object() 函数允许在对象中使用 XML 解析器。

语法

```
xml_set_object(parser,object)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
object	必需。规定设置解析器的对象。

说明

该函数使得 *parser* 指定的解析器可以被用在 *object* 对象中。所有的回叫函数（callback function）都可以由 xml*set_element_handler()* 等函数来设置，它们被假定为 *_object* 对象的方法。

例子

```
<?php
class XMLParser
{
var $xmlparser;

function XMLParser()
  {
  $this->xmlparser = xml_parser_create();
  xml_set_object($this->xmlparser, $this);
  xml_set_character_data_handler($this->xmlparser,"char");
  xml_set_element_handler($this->xmlparser, "start_tag","end_tag");
  }

function parse($data)
  {
  xml_parse($this->xmlparser, $data);
  }

function parse_File($xmlfile)
  {
  $fp = fopen($xmlfile, 'r');
  while ($xmldata = fread($fp, 4096))
    {
    if
    (!xml_parse($this->xmlparser, $xmldata))
      {
      //If error
      die( print "ERROR: "
      . xml_error_string(xml_get_error_code($this->xmlparser))
      . "<br />Line: "
      . xml_get_current_line_number($this->xmlparser)
      . "<br />Column: "
      . xml_get_current_column_number($this->xmlparser)
      . "<br />");
      }
    }
  }

function start_tag($xmlparser, $tag, $attributes)
  {
  print $tag . "<br />";
  }

function end_tag(){}

function char($xmlparser,$data)
  {
  echo $data . "<br />";
  }

function close_Parser()
  {
  xml_parser_free($this->xmlparser);
  }
}

$myxmlparser = new XMLParser();
$myxmlparser->parse_File("test.xml");
$myxmlparser->close_parser();

?>
```

PHP xml_set_processing_instruction_handler() 函数

定义和用法

xml_set_processing_instruction_handler() 函数规定当解析器在 XML 文档中找到处理指令时所调用的函数。

处理指令包含在 <? 和 ?> 分隔符中。

如果处理器被成功的建立，该函数将返回 true；否则返回 false。

例子：在本例中，处理指令把一个样式表与 XML 文档关联起来：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="default.xsl" type="text/xml"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

语法

```
xml_set_processing_instruction_handler(parser,handler)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
handler	必需。规定一个函数。

由 *handler* 参数规定的函数必须有三个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
target	必需。规定包含处理指令目标的变量。
data	必需。规定包含处理指令数据的变量。

说明

handler 参数也可以是一个数组，其中包含对象引用和方法名。

例子

```
<?php

$parser=xml_parser_create();

function char($parser,$data)
  {
  echo $data;
  }

function pi_handler($parser, $target, $data)
  {
  echo "Target: $target<br />";
  echo "Data: $data<br />";
  }

xml_set_character_data_handler($parser,"char");
xml_set_processing_instruction_handler($parser, "pi_handler");
$fp=fopen("test.xml","r");

while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

xml_parser_free($parser);

?>
```

PHP xml_set_unparsed_entity_decl_handler() 函数

定义和用法

xml_set_unparsed_entity_decl_handler() 函数规定在遇到无法解析的实体名称（NDATA）声明时被调用的函数。

如果处理器被成功的建立，该函数将返回 true；否则返回 false。

语法

```
xml_set_unparsed_entity_decl_handler(parser,handler)
```

参数	描述
parser	必需。规定要使用的 XML 解析器。
handler	必需。规定一个函数。

由 *handler* 参数规定的函数必须有六个参数：

参数	描述
parser	必需。规定一个变量，包含调用处理器的 XML 解析器。
name	必需。规定包含实体名称的变量。
base	必需。规定一个变量，包含解析实体的系统标识符（system_id ）的基础。当前该参数通常都被设置为空字符串。
system_id	必需。规定包含实体的系统标识符的变量。
public_id	必需。规定包含实体的公共标识符的变量。
notation	必需。规定一个变量，包含标识实体数据类型的符号。

说明

handler 参数也可以是一个数组，其中包含对象引用和方法名。

例子

```
<?php

$parser=xml_parser_create();

function char($parser,$data)
  {
  echo $data;
  }

function unparsed_ent_handler($parser,$entname,
$base,$sysID,$pubID,$notname)
  {
  print "$entname";
  print "$sysID";
  print "$pubID";
  print "$notname";
  }

xml_set_character_data_handler($parser,"char");
xml_set_unparsed_entity_decl_handler($parser,
"unparsed_ent_handler");

$fp=fopen("test.xml","r");

while ($data=fread($fp,4096))
  {
  xml_parse($parser,$data,feof($fp)) or
  die (sprintf("XML Error: %s at line %d",
  xml_error_string(xml_get_error_code($parser)),
  xml_get_current_line_number($parser)));
  }

xml_parser_free($parser);

?>
```

PHP Zip File 函数

PHP Zip File 简介

压缩文件函数允许我们读取压缩文件。

安装

如需在服务器上运行 Zip File 函数，必须安装这些库：

- Guido Draheim 的 ZZIPlib 库：下载 ZZIPlib 库
- Zip PELC 扩展：下载 Zip PELC 扩展

在 Linux 系统上安装

PHP 5+：_*Zip* 函数和 *Zip* 库默认不会启用，必须从上面的链接下载。请使用 --with-zip=DIR_ 配置选项来包含 Zip 支持。

在 Windows 系统上安装

_PHP 5+：_Zip 函数默认不会启用，必须从上面的链接下载 php_zip.dll 和 ZZIPlib 库。必须在php.ini 之内启用 php_zip.dll。

如需启用任何 PHP 扩展，PHP extension_dir 设置（在 php.ini 文件中）应该设置为该 PHP 扩展所在的目录。举例 extension_dir 的值可能是 c:\php\ext。

PHP Zip File 函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
zip_close()	关闭 ZIP 文件。	4
zip_entry_close()	关闭 ZIP 文件中的一个项目。	4
zip_entry_compressedsize()	返回 ZIP 文件中的一个项目的被压缩尺寸。	4
zip_entry_compressionmethod()	返回 ZIP 文件中的一个项目的压缩方法。	4
zip_entry_filesize()	返回 ZIP 文件中的一个项目的实际文件尺寸。	4
zip_entry_name()	返回 ZIP 文件中的一个项目的名称。	4
zip_entry_open()	打开 ZIP 文件中的一个项目以供读取。	4
zip_entry_read()	读取 ZIP 文件中的一个打开的项目。	4
zip_open()	打开 ZIP 文件。	4
zip_read()	读取 ZIP 文件中的下一个项目。	4

PHP Zip File 常量

无。

PHP zip_close() 函数

定义和用法

zip_close() 函数关闭由 zip_open() 函数打开的 zip 档案文件。

语法

```
zip_close(zip)
```

参数	描述
zip	必需。规定要关闭的 zip 资源（由 zip_open() 打开的 zip 文件）。

例子

```
<?php
$zip = zip_open("test.zip");

zip_read($zip);

// 一些代码...

zip_close($zip);
?>
```

PHP zip_entry_close() 函数

定义和用法

zip_entry_close() 函数关闭由 zip_entry_open() 函数打开的 zip 档案文件。

语法

```
zip_entry_close(zip_entry)
```

参数	描述
zip_entry	必需。规定要关闭的 zip 项目资源（由 zip_read() 打开的 zip 项目）。

例子

```
<?php
$zip = zip_open("test.zip");

if ($zip)
  {
  while ($zip_entry = zip_read($zip))
    {
    echo "<p>";
    echo "Name: " . zip_entry_name($zip_entry) . "<br />";

    if (zip_entry_open($zip, $zip_entry))
      {
      // 一些代码...
      zip_entry_close($zip_entry);
      }
    echo "</p>";
  }
zip_close($zip);
}
?>
```

PHP zip_entry_compressedsize() 函数

定义和用法

zip_entry_compressedsize() 函数返回 zip 档案项目的压缩文件尺寸。

语法

```
zip_entry_compressedsize(zip_entry)
```

参数	描述
zip_entry	必需。规定要读取的 zip 项目资源（由 zip_read() 打开的 zip 项目）。

例子

```
<?php
$zip = zip_open("test.zip");

if ($zip)
  {
  while ($zip_entry = zip_read($zip))
    {
    echo "<p>";
    echo "Name: " . zip_entry_name($zip_entry) . "<br />";
    echo "Compressed Size: "
    . zip_entry_compressedsize($zip_entry);
    echo "</p>";
    }
  zip_close($zip);
  }
?>
```

输出：

```
Name: ziptest.txt
Compressed Size: 68

Name: htmlziptest.html
Compressed Size: 159
```

PHP zip_entry_compressionmethod() 函数

定义和用法

zip_entry_compressionmethod() 函数返回 zip 档案项目的压缩方法。

语法

```
zip_entry_compressionmethod(zip_entry)
```

参数	描述
zip_entry	必需。规定要读取的 zip 项目资源（由 zip_read() 打开的 zip 项目）。

例子

```
<?php
$zip = zip_open("test.zip");

if ($zip)
  {
  while ($zip_entry = zip_read($zip))
    {
    echo "<p>";
    echo "Name: " . zip_entry_name($zip_entry) . "<br />";
    echo "Compression Method: "
    . zip_entry_compressionmethod($zip_entry);
    echo "</p>";
    }
  zip_close($zip);
  }
?>
```

输出：

```
Name: ziptest.txt
Compression Method: deflated

Name: htmlziptest.html
Compression Method: deflated
```

PHP zip_entry_filesize() 函数

定义和用法

zip_entry_filesize() 函数返回 zip 档案项目的原始大小（在压缩之前）。

语法

```
zip_entry_filesize(zip_entry)
```

参数	描述
zip_entry	必需。规定要读取的 zip 项目资源（由 zip_read() 打开的 zip 项目）。

例子

```
<?php
$zip = zip_open("test.zip");

if ($zip)
  {
  while ($zip_entry = zip_read($zip))
    {
    echo "<p>";
    echo "Name: " . zip_entry_name($zip_entry) . "<br />";
    echo "Original size: " . zip_entry_filesize($zip_entry);
    echo "</p>";
    }
  zip_close($zip);
  }
?>
```

输出：

```
Name: ziptest.txt
Original size: 68

Name: htmlziptest.html
Original size: 159
```

PHP zip_entry_name() 函数

定义和用法

zip_entry_name() 函数返回 zip 档案项目的名称。

语法

```
zip_entry_name(zip_entry)
```

参数	描述
zip_entry	必需。规定要读取的 zip 项目资源（由 zip_read() 打开的 zip 项目）。

例子

```
<?php
$zip = zip_open("test.zip");

if ($zip)
  {
  while ($zip_entry = zip_read($zip))
    {
    echo "Name: " . zip_entry_name($zip_entry) . "<br />";
    }
  zip_close($zip);
  }
?>
```

输出类似：

```
Name: ziptest.txt
Name: htmlziptest.html
```

PHP zip_entry_open() 函数

定义和用法

zip_entry_open() 函数打开一个 ZIP 档案项目以供读取。

语法

```
zip_entry_open(zip,zip_entry,mode)
```

参数	描述
zip	必需。规定要读取的 zip 资源（由 zip_open() 打开的 zip 文件）。
zip_entry	必需。规定要打开的 zip 项目资源（由 zip_read() 打开的 zip 项目）。
mode	可选。规定 zip 档案项目的访问类型。

说明

在 PHP 5 中，mode 会被忽略，且总为 "rb"。这是因为在 PHP 中的 zip 支持是只读的。

例子

```
<?php
$zip = zip_open("test.zip");

if ($zip)
  {
  while ($zip_entry = zip_read($zip))
    {
    echo "<p>";
    echo "Name: " . zip_entry_name($zip_entry) . "<br />";

    if (zip_entry_open($zip, $zip_entry))
      {
      // some code
      }
    echo "</p>";
  }

zip_close($zip);

}
?>
```

PHP zip_entry_read() 函数

定义和用法

zip_entry_read() 函数从打开的 zip 档案项目中获取内容。

如果成功，则返回项目的内容。如果失败，则返回 false。

语法

```
zip_entry_read(zip_entry,length)
```

参数	描述
zip_entry	必需。规定要读取的 zip 项目资源（由 zip_read() 打开的 zip 项目）。
length	可选。规定返回的字节数。默认是 1024。

例子

```
<?php
$zip = zip_open("test.zip");

if ($zip)
  {
  while ($zip_entry = zip_read($zip))
    {
    echo "<p>";
    echo "Name: " . zip_entry_name($zip_entry) . "<br />";

    if (zip_entry_open($zip, $zip_entry))
      {
      echo "File Contents:<br/>";
      $contents = zip_entry_read($zip_entry);
      echo "$contents<br />";
      zip_entry_close($zip_entry);
      }
    echo "</p>";
  }

zip_close($zip);

}
?>
```

PHP zip_open() 函数

定义和用法

zip_open() 函数打开 ZIP 文件以供读取。

如果成功，则返回 zip 文件档案资源。如果失败，则返回 false。

语法

```
zip_open(filename)
```

参数	描述
filename	必需。规定要打开的 zip 文件的文件名和路径。

提示和注释

提示：新打开的 zip 文件资源之后可被 zip_read() 和 zip_close() 函数使用。

例子

```
<?php
$zip = zip_open("test.zip");

zip_read($zip);

// 一些代码...

zip_close($zip);
?>
```

PHP zip_read() 函数

定义和用法

zip_read() 函数读取打开的 zip 档案中的下一个文件。

如果成功，则返回包含 zip 档案中一个文件的资源。如果没有更多的项目可供读取，则返回 false。

语法

```
zip_read(zip)
```

参数	描述
zip	必需。规定要读取的 zip 资源（由 zip_open() 打开的 zip 文件）。

提示和注释

提示：由 zip_read() 函数返回的资源可供 zip_entry... 类的函数使用。

例子

```
<?php
$zip = zip_open("test.zip");

zip_read($zip);

// 一些代码...

zip_close($zip);
?>
```

PHP 杂项函数

PHP 杂项函数简介

我们把不属于其他类别的函数归纳到这个页面。

安装

杂项函数是 PHP 核心的组成部分。无需安装即可使用这些函数。

Runtime 配置

杂项函数函数的行为受到 php.ini 中设置的影响。

杂项函数配置选项：

名称	默认	描述	可更改
ignore_user_abort	"0"	FALSE 指示只要脚本在客户机终止连接后尝试进行输出，脚本将被终止。	PHP_INI_ALL
highlight.string	"#DD0000"	供突出显示符合 PHP 语法的字符串而使用的颜色。	PHP_INI_ALL
highlight.comment	"#FF8000"	供突出显示 PHP 注释而使用的颜色。	PHP_INI_ALL
highlight.keyword	"#007700"	供突出显示 PHP 关键词而使用的颜色（比如圆括号和分号）。	PHP_INI_ALL
highlight.bg	"#FFFFFF"	背景颜色。	PHP_INI_ALL
highlight.default	"#0000BB"	PHP 语法的默认颜色。	PHP_INI_ALL
highlight.html	"#000000"	HTML 代码的颜色。	PHP_INI_ALL
browscap	NULL	浏览器性能文件的名称和位置（例如：browscap.ini）。	PHP_INI_SYSTEM

PHP 杂项函数

PHP：指示支持该函数的最早的 PHP 版本。

函数	描述	PHP
connection_aborted()	检查是否断开客户机。	3
connection_status()	返回当前的连接状态。	3
connection_timeout()	在 PHP 4.0.5 中不赞成使用。	3
constant()	返回一个常量的值。	4
define()	定义一个常量。	3
defined()	检查某常量是否存在。	3
die()	输出一条消息，并退出当前脚本。	3
eval()	把字符串按照 PHP 代码来计算。	3
exit()	输出一条消息，并退出当前脚本。	3
get_browser()	返回用户浏览器的性能。	3
highlight_file()	对文件进行语法高亮显示。	4
highlight_string()	对字符串进行语法高亮显示。	4
ignore_user_abort()	设置与客户机断开是否会终止脚本的执行。	3
pack()	把数据装入一个二进制字符串。	3
php_check_syntax()	在 PHP 5.0.5 中不赞成使用。	5
php_strip_whitespace()	返回已删除 PHP 注释以及空白字符的源代码文件。	5
show_source()	highlight_file() 的别名。	4
sleep()	延迟代码执行若干秒。	3
time_nanosleep()	延迟代码执行若干秒和纳秒。	5
time_sleep_until()	延迟代码执行指定的时间。	5
uniqid()	生成唯一的 ID。	3
unpack()	从二进制字符串对数据进行解包。	3
usleep()	延迟代码执行若干微秒。	3

PHP Date / Time 常量

PHP：指示支持该常量的最早的 PHP 版本。

常量	描述	PHP
CONNECTION_ABORTED		
CONNECTION_NORMAL		
CONNECTION_TIMEOUT		
COMPILER_HALT_OFFSET	5	

PHP connection_aborted() 函数

定义和用法

connection_aborted() 函数检查是否断开客户机。

如果已终止连接，则该函返回 1，否则返回 0。

语法

```
connection_aborted()
```

例子

创建一个函数，在客户机终止脚本时写入一条日志消息：

```
<?php
function check_abort()
  {
  if (connection_aborted())
  error_log ("Script $GLOBALS[SCRIPT_NAME]" .
  "$GLOBALS[SERVER_NAME] was aborted by the user.");
  }

//要执行的一些代码

// 在脚本结束时调用 check_abort 函数
register_shutdown_function("check_abort");
?>
```

PHP connection_status() 函数

定义和用法

connection_status() 函数返回当前的连接状态。

可返回的可能值：

- 0 - CONNECTION_NORMAL - 连接运行正常
- 1 - CONNECTION_ABORTED - 连接由用户或网络错误终止
- 2 - CONNECTION_TIMEOUT - 连接超时
- 3 - CONNECTION_ABORTED & CONNECTION_TIMEOUT

语法

```
connection_status()
```

例子

```
<?php

switch (connection_status())
{
case CONNECTION_NORMAL:
   $txt = 'Connection is in a normal state';
   break;
case CONNECTION_ABORTED:
   $txt = 'Connection aborted';
   break;
case CONNECTION_TIMEOUT:
   $txt = 'Connection timed out';
   break;
case (CONNECTION_ABORTED & CONNECTION_TIMEOUT):
   $txt = 'Connection aborted and timed out';
   break;
default:
   $txt = 'Unknown';
   break;
}

echo $txt;
?>
```

PHP constant() 函数

定义和用法

constant() 函数返回常量的值。

语法

```
constant(constant)
```

参数	描述
constant	必需。规定要检查的常量的名称。

提示和注释

注释：该函数仅适用于 class 常量。

例子

```
<?php
//定义一个常量
define("GREETING","Hello world!");

echo constant("GREETING");
?>
```

输出：

```
Hello world!
```

PHP define() 函数

定义和用法

define() 函数定义一个常量。

常量类似变量，不同之处在于：

- 在设定以后，常量的值无法更改
- 常量名不需要开头的美元符号 ($)
- 作用域不影响对常量的访问
- 常量值只能是字符串或数字

语法

```
define(_name_,_value_,_case_insensitive_)
```

参数	描述
name	必需。规定常量的名称。
value	必需。规定常量的值。
case_insensitive	可选。规定常量的名称是否对大小写敏感。若设置为 true，则对大小写不敏感。默认是 false（大小写敏感）。

例子

例子 1

定义一个大小写敏感的常量：

```
<?php
define("GREETING","Hello world!");
echo constant("GREETING");
?>
```

输出：

```
Hello world!
```

例子 2

定义一个大小写不敏感的常量：

```
<?php
define("GREETING","Hello world!",TRUE);
echo constant("greeting");
?>
```

输出：

```
Hello world!
```

PHP defined() 函数

定义和用法

defined() 函数检查某常量是否存在。

若常量存在，则返回 true，否则返回 false。

语法

```
defined(name)
```

参数	描述
name	必需。规定要检查的常量的名称。

例子

```
<?php
define("GREETING","Hello world!");
echo defined("GREETING");
?>
```

输出：

```
1
```

PHP die() 函数

定义和用法

die() 函数输出一条消息，并退出当前脚本。

该函数是 exit() 函数的别名。

语法

```
die(status)
```

参数	描述
status	必需。规定在退出脚本之前写入的消息或状态号。状态号不会被写入输出。

说明

如果 *status* 是字符串，则该函数会在退出前输出字符串。

如果 *status* 是整数，这个值会被用作退出状态。退出状态的值在 0 至 254 之间。退出状态 255 由 PHP 保留，不会被使用。状态 0 用于成功地终止程序。

提示和注释

注释：如果 PHP 的版本号大于等于 4.2.0，那么在 *status* 是整数的情况下，不会输出该参数。

例子

```
<?php
$site = "http://www.w3school.com.cn/";
fopen($site,"r")
or die("Unable to connect to $site");
?>
```

PHP eval() 函数

定义和用法

eval() 函数把字符串按照 PHP 代码来计算。

该字符串必须是合法的 PHP 代码，且必须以分号结尾。

如果没有在代码字符串中调用 return 语句，则返回 NULL。如果代码中存在解析错误，则 eval() 函数返回 false。

语法

```
eval(phpcode)
```

参数	描述
phpcode	必需。规定要计算的 PHP 代码。

提示和注释

注释：返回语句会立即终止对字符串的计算。

注释：该函数对于在数据库文本字段中供日后计算而进行的代码存储很有用。

例子

```
<?php
$string = "beautiful";
$time = "winter";

$str = 'This is a $string $time morning!';
echo $str. "<br />";

eval("\$str = \"$str\";");
echo $str;
?>
```

输出：

```
This is a $string $time morning!
This is a beautiful winter morning!
```

PHP exit() 函数

定义和用法

exit() 函数输出一条消息，并退出当前脚本。

该函数是 die() 函数的别名。

语法

```
exit(status)
```

参数	描述
status	必需。规定在退出脚本之前写入的消息或状态号。状态号不会被写入输出。

说明

如果 *status* 是字符串，则该函数会在退出前输出字符串。

如果 *status* 是整数，这个值会被用作退出状态。退出状态的值在 0 至 254 之间。退出状态 255 由 PHP 保留，不会被使用。状态 0 用于成功地终止程序。

提示和注释

注释：如果 PHP 的版本号大于等于 4.2.0，那么在 *status* 是整数的情况下，不会输出该参数。

例子

```
<?php
$site = "http://www.w3school.com.cn/";
fopen($site,"r")
or exit("Unable to connect to $site");
?>
```

PHP get_browser() 函数

定义和用法

get_browser() 函数返回用户浏览器的性能。

该函数通过查阅用户的 browscap.ini 文件，来测定用户浏览器的性能。

若成功，则该函数返回包含用户浏览器信息的一个对象或一个数组，若失败，则返回 false。

语法

```
get_browser(user_agent,return_array)
```

参数	描述
user_agent	可选。规定 HTTP 用户代理的名称。默认是 $HTTP_USER_AGENT 的值。您可以通过设置 NULL 绕过该参数。
return_array	可选。如果该参数设置为 true，本函数会返回一个数组而不是对象。

提示和注释

注释：返回语句会立即终止对字符串的计算。

注释：该函数对于在数据库文本字段中供日后计算而进行的代码存储很有用。

例子

```
<?php
echo $_SERVER['HTTP_USER_AGENT'] . "<br /><br />";
$browser = get_browser(null,true);
print_r($browser);
?>
```

输出：

```
Mozilla/4.0
(compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)

Array
(
[browser_name_regex] => ^mozilla/.\.0
(compatible; msie 6\.0.*;.*windows nt 5\.1.*\.net clr.*).*$
[browser_name_pattern] => Mozilla/?.0
(compatible; MSIE 6.0*;*Windows NT 5.1*.NET CLR*)*
[parent] => IE 6.0
[platform] => WinXP
[netclr] => 1
[browser] => IE
[version] => 6.0
[majorver] => 6
[minorver] => 0
[css] => 2
[frames] => 1
[iframes] => 1
[tables] => 1
[cookies] => 1
[backgroundsounds] => 1
[vbscript] => 1
[javascript] => 1
[javaapplets] => 1
[activexcontrols] => 1
[cdf] => 1
[aol] =>
[beta] =>
[win16] =>
[crawler] =>
[stripper] =>
[wap] =>
[ak] =>
[sk] =>
)
```

PHP highlight_file() 函数

定义和用法

highlight_file() 函数对文件进行语法高亮显示。

语法

```
highlight_file(filename,return)
```

参数	描述
filename	必需。要进行高亮处理的 PHP 文件的路径。
return	可选。如果设置 true，则本函数返回高亮处理的代码。

说明

本函数通过使用 PHP 语法高亮程序中定义的颜色，输出或返回包含在 *filename* 中的代码的语法高亮版本。

许多服务器被配置为对带有 *phps* 后缀的文件进行自动高亮处理。例如，在查看 example.phps 时，将显示该文件被语法高亮显示的源代码。要启用该功能，请把下面这一行添加到 *httpd.conf*：

```
AddType application/x-httpd-php-source .phps
```

返回值

如果 *return* 参数被设置为 true，那么该函数会返回被高亮处理的代码，而不是输出它们。否则，若成功，则返回 true，失败则返回 false。

提示和注释

警告：需要注意的是，在使用 highlight_file() 函数时，请不要因为疏忽而泄露诸如密码或其他类型的敏感信息，否则会出现潜在的安全风险。

例子

"test.php":

```
<html>
<body>
<?php
highlight_file("test.php");
?>
</body>
</html>
```

输出：

```
<html>
<body>
<?php
highlight_file("test.php");
?>
</body>
</html>
```

在浏览器中查看的结果类似这样：

```
<html>
<body>
<code>
<span style="color: #000000">&lt;html&gt;
<br />
&lt;body&gt;
<br />
<span style="color: #0000BB">&lt;?php
<br />highlight_file</span>
<span style="color: #007700">(</span>
<span style="color: #DD0000">"test.php"</span>
<span style="color: #007700">);<br /></span>
<span style="color: #0000BB">?&gt;<br /></span>
&lt;/body&gt;
<br />
&lt;/html&gt;</span>
</code>
</body>
</html>
```

PHP highlight_string() 函数

定义和用法

highlight_string() 函数对字符串进行语法高亮显示。

语法

```
highlight_string(string,return)
```

参数	描述
string	必需。要进行高亮处理的字符串。
return	可选。如果设置 true，则本函数返回高亮处理的代码。

说明

本函数通过使用 PHP 语法高亮程序中定义的颜色，输出或返回给定的 PHP 代码的语法高亮版本。

返回值

如果 *return* 参数被设置为 true，那么该函数会以字符串返回被高亮处理的代码，而不是输出它们。否则，若成功，则返回 true，失败则返回 false。

例子

```
<html>
<body>
<?php
highlight_string("Hello world! <?php phpinfo(); ?>");
?>
</body>
</html>
```

输出：

```
Hello world! <?php phpinfo();?>
```

在浏览器中查看的结果类似这样：

```
<html>
<body>
 <code>
  <span style="color: #000000">Hello world! 
  <span style="color: #0000BB">&lt;?php phpinfo</span>
  <span style="color: #007700">();</span>
  <span style="color: #0000BB">?&gt;</span>
  </span>
 </code>
</body>
</html>
```

PHP ignore_user_abort() 函数

定义和用法

ignore_user_abort() 函数设置与客户机断开是否会终止脚本的执行。

本函数返回 user-abort 设置的之前的值（一个布尔值）。

语法

```
ignore_user_abort(setting)
```

参数	描述
setting	可选。如果设置为 true，则忽略与用户的断开，如果设置为 false，会导致脚本停止运行。如果未设置该参数，会返回当前的设置。

提示和注释

注释：PHP 不会检测到用户是否已断开连接，直到尝试向客户机发送信息为止。简单地使用 echo 语句无法确保信息发送，参阅 flush() 函数。

例子

```
<?php
ignore_user_abort();
?>
```

输出：

```
0
```

PHP pack() 函数

定义和用法

pack() 函数把数据装入一个二进制字符串。

语法

```
pack(format,args+)
```

参数	描述
format	必需。规定在包装数据时所使用的格式。
args+	可选。规定被包装的一个或多个参数。

format 参数的可能值：

- a - NUL-padded string
- A - SPACE-padded string
- h - Hex string, low nibble first
- H - Hex string, high nibble first
- c - signed char
- C - unsigned char
- s - signed short (always 16 bit, machine byte order)
- S - unsigned short (always 16 bit, machine byte order)
- n - unsigned short (always 16 bit, big endian byte order)
- v - unsigned short (always 16 bit, little endian byte order)
- i - signed integer (machine dependent size and byte order)
- I - unsigned integer (machine dependent size and byte order)
- l - signed long (always 32 bit, machine byte order)
- L - unsigned long (always 32 bit, machine byte order)
- N - unsigned long (always 32 bit, big endian byte order)
- V - unsigned long (always 32 bit, little endian byte order)
- f - float (machine dependent size and representation)
- d - double (machine dependent size and representation)
- x - NUL byte
- X - Back up one byte
- @ - NUL-fill to absolute position

例子

例子 1

```
<?php
echo pack("C3",80,72,80);
?>
```

输出：

```
PHP
```

例子 2

```
<?php
echo pack("C*",80,72,80);
?>
```

输出：

```
PHP
```

PHP strip_whitespace() 函数

定义和用法

strip_whitespace() 函数返回已删除 PHP 注释以及空白字符的源代码文件。

该函数对于检测脚本中的实际代码量很有用。

语法

```
strip_whitespace(filename)
```

参数	描述
filename	必需。规定文件名。

说明

若成功，则返回被剥离的源代码，若失败，则返回空字符串。

注释：在 PHP 5.0.1，该函数的行为与上面的描述一致。在这之前，它仅返回空字符串。

例子

"test.php":

```
<?php
// PHP comment

/*
*  Another PHP comment
*/

echo php_strip_whitespace      ("test.php");
?>
```

输出：

```
<?php
echo php_strip_whitespace ("test.php"); ?>
```

PHP show_source() 函数

定义和用法

show_source() 函数对文件进行语法高亮显示。

本函数是 highlight_file() 的别名。

语法

```
show_source(filename,return)
```

参数	描述
filename	必需。要进行高亮处理的 PHP 文件的路径。
return	可选。如果设置 true，则本函数返回高亮处理的代码。

说明

本函数通过使用 PHP 语法高亮程序中定义的颜色，输出或返回包含在 *filename* 中的代码的语法高亮版本。

许多服务器被配置为对带有 *phps* 后缀的文件进行自动高亮处理。例如，在查看 example.phps 时，将显示该文件被语法高亮显示的源代码。要启用该功能，请把下面这一行添加到 *httpd.conf*：

```
AddType application/x-httpd-php-source .phps
```

返回值

如果 *return* 参数被设置为 true，那么该函数会返回被高亮处理的代码，而不是输出它们。否则，若成功，则返回 true，失败则返回 false。

提示和注释

警告：需要注意的是，在使用 show_source() 函数时，请不要因为疏忽而泄露诸如密码或其他类型的敏感信息，否则会出现潜在的安全风险。

例子

"test.php":

```
<html>
<body>
<?php
show_source("test.php");
?>
</body>
</html>
```

输出：

```
<html>
<body>
<?php
show_source("test.php");
?>
</body>
</html>
```

在浏览器中查看的结果类似这样：

```
<html>
<body>
<code>
<span style="color: #000000">&lt;html&gt;
<br />
&lt;body&gt;
<br />
<span style="color: #0000BB">&lt;?php
<br />show_source</span>
<span style="color: #007700">(</span>
<span style="color: #DD0000">"test.php"</span>
<span style="color: #007700">);<br /></span>
<span style="color: #0000BB">?&gt;<br /></span>
&lt;/body&gt;
<br />
&lt;/html&gt;</span>
</code>
</body>
</html>
```

PHP sleep() 函数

定义和用法

sleep() 函数延迟代码执行若干秒。

语法

```
sleep(seconds)
```

参数	描述
seconds	必需。以秒计的暂停时间。

返回值

若成功，返回 0，否则返回 false。

错误／异常

如果指定的描述 seconds 是负数，该函数将生成一个 E_WARNING。

例子

```
<?php
echo date('h:i:s') . "<br />";

//暂停 10 秒
sleep(10);

//重新开始
echo date('h:i:s');

?>
```

输出：

```
12:00:08
12:00:18
```

PHP time_nanosleep() 函数

定义和用法

time_nanosleep() 函数延迟代码执行若干秒和纳秒。

语法

```
time_nanosleep(seconds,nanoseconds)
```

参数	描述
seconds	必需。必须是正整数。
nanoseconds	必需。必须是小于 10 亿的正整数。

说明

延迟程序执行指定的 *seconds* 和 *nanoseconds* 数。

返回值

如果成功则返回 TRUE，失败则返回 FALSE

如果延迟被一个信号中断，将返回带有以下组件的关联数组：

- seconds - 延迟中剩余的秒数
- nanoseconds - 延迟中剩余的纳秒数

提示和注释

注释：本函数未在 Windows 平台下实现。

例子

```
<?php
if (time_nanosleep(3,500000000) === true)
  {
  echo "暂停 3 秒半";
  }
?>
```

PHP time_sleep_until() 函数

定义和用法

time_sleep_until() 函数延迟代码执行直到指定的时间。

语法

```
time_sleep_until(timestamp)
```

参数	描述
timestamp	必需。脚本唤醒时的时间戳。

说明

使脚本暂停执行，直到指定的 *timestamp*。

返回值

如果成功则返回 TRUE，失败则返回 FALSE。

错误／异常

如果指定的时间戳位于过去，则该函数将生成一个 E_WARNING。

提示和注释

注释：所有信号都将在脚本唤醒后递送。

注释：本函数未在 Windows 平台下实现。

例子

```
<?php
// 从现在起 10 秒后唤醒
time_sleep_until(time()+10);
?>
```

PHP uniqid() 函数

定义和用法

uniqid() 函数基于以微秒计的当前时间，生成一个唯一的 ID。

语法

```
uniqid(prefix,more_entropy)
```

参数	描述
prefix	可选。为 ID 规定前缀。如果两个脚本恰好在相同的微秒生成 ID，该参数很有用。
more_entropy	可选。规定位于返回值末尾的更多的熵。

说明

如果 *prefix* 参数为空，则返回的字符串有 13 个字符串长。如果 *more_entropy* 参数设置为 true，则是 23 个字符串长。

如果 *more_entropy* 参数设置为 true，则在返回值的末尾添加额外的熵（使用组合线形同余数生成程序），这样可以结果的唯一性更好。

返回值

以字符串的形式返回唯一标识符。

提示和注释

注释：由于基于系统时间，通过该函数生成的 ID 不是最佳的。如需生成绝对唯一的 ID，请使用 md5() 函数（请在字符串函数参考中查找）。

例子

```
<?php
echo uniqid();
?>
```

输出类似：

```
4415297e3af8c
```

PHP unpack() 函数

定义和用法

unpack() 函数从二进制字符串对数据进行解包。

语法

```
unpack(format,data)
```

参数	描述
format	必需。规定在解包数据时所使用的格式。
data	可选。规定被解包的二进制数据。

format 参数的可能值：

- a - NUL-padded string
- A - SPACE-padded string
- h - Hex string, low nibble first
- H - Hex string, high nibble first
- c - signed char
- C - unsigned char
- s - signed short (always 16 bit, machine byte order)
- S - unsigned short (always 16 bit, machine byte order)
- n - unsigned short (always 16 bit, big endian byte order)
- v - unsigned short (always 16 bit, little endian byte order)
- i - signed integer (machine dependent size and byte order)
- I - unsigned integer (machine dependent size and byte order)
- l - signed long (always 32 bit, machine byte order)
- L - unsigned long (always 32 bit, machine byte order)
- N - unsigned long (always 32 bit, big endian byte order)
- V - unsigned long (always 32 bit, little endian byte order)
- f - float (machine dependent size and representation)
- d - double (machine dependent size and representation)
- x - NUL byte
- X - Back up one byte
- @ - NUL-fill to absolute position

例子

例子 1

```
<?php
$data = "PHP";
print_r(unpack("C*",$data));
?>
```

输出：

```
Array
(
[1] => 80
[2] => 72
[3] => 80
)
```

例子 2

```
<?php
$data = "PHP";
print_r(unpack("C*myint",$data));
?>
```

输出：

```
Array
(
[myint1] => 80
[myint2] => 72
[myint3] => 80
)
```

例子 3

```
<?php
$bin = pack("c2n2",0x1234,0x5678,65,66);
print_r(unpack("c2chars/n2int",$bin));
?>
```

输出：

```
Array
(
[chars1] => 52
[chars2] => 120
[int1] => 65
[int2] => 66
)
```

PHP usleep() 函数

定义和用法

usleep() 函数延迟代码执行若干微秒。

语法

```
usleep(microseconds)
```

参数	描述
microseconds	必需。以微秒计的暂停时间。

返回值

无返回值。

提示和注释

注释：在 PHP 5 之前，该函数无法工作于 Windows 系统上。

注释：一微秒等于百万分之一秒。

例子

```
<?php
echo date('h:i:s') . "<br />";

//延迟 10 描述
usleep(10000000);

//再次开始
echo date('h:i:s');
?>
```

输出：

```
09:23:14
09:23:24
```

PHP 5 时区

PHP 支持的时区

下面是 PHP 支持的时区的完整列表，这些对一些 PHP 日期函数很有用。

- 非洲
- 美洲
- 南极洲
- 北冰洋
- 亚洲
- 大西洋
- 大洋洲
- 欧洲
- 印度洋
- 太平洋

非洲

Africa/Abidjan	Africa/Accra	Africa/Addis_Ababa	Africa/Algiers
Africa/Asmera	Africa/Bamako	Africa/Bangui	Africa/Banjul
Africa/Blantyre	Africa/Brazzaville	Africa/Bujumbura	Africa/Cairo
Africa/Ceuta	Africa/Conakry	Africa/Dakar	Africa/Dar_es_Sala
Africa/Douala	Africa/El_Aaiun	Africa/Freetown	Africa/Gaborone
Africa/Johannesburg	Africa/Juba	Africa/Kampala	Africa/Khartoum
Africa/Kinshasa	Africa/Lagos	Africa/Libreville	Africa/Lome
Africa/Lubumbashi	Africa/Lusaka	Africa/Malabo	Africa/Maputo
Africa/Mbabane	Africa/Mogadishu	Africa/Monrovia	Africa/Nairobi
Africa/Niamey	Africa/Nouakchott	Africa/Ouagadougou	Africa/Porto-Novo
Africa/Timbuktu	Africa/Tripoli	Africa/Tunis	Africa/Windhoek

美洲

America/Adak	America/Anchorage	Ame

America/Antigua	America/Araguaina	America/Arge
America/Argentina/Catamarca	America/Argentina/ComodRivadavia	America/Arge
America/Argentina/Jujuy	America/Argentina/La_Rioja	America/Arge
America/Argentina/Rio_Gallegos	America/Argentina/Salta	America/Arge
America/Argentina/San_Luis	America/Argentina/Tucuman	America/Arge
America/Aruba	America/Asuncion	America/Atiko
America/Atka	America/Bahia	America/Bahi
America/Barbados	America/Belem	America/Beliz
America/Blanc-Sablon	America/Boa_Vista	America/Bogc
America/Boise	America/Buenos_Aires	America/Cam
America/Campo_Grande	America/Cancun	America/Cara
America/Catamarca	America/Cayenne	America/Cayr
America/Chicago	America/Chihuahua	America/Cora
America/Cordoba	America/Costa_Rica	America/Cres
America/Cuiaba	America/Curacao	America/Danr
America/Dawson	America/Dawson_Creek	America/Denv
America/Detroit	America/Dominica	America/Edm
America/Eirunepe	America/El_Salvador	America/Ense
America/Fort_Wayne	America/Fortaleza	America/Glac
America/Godthab	America/Goose_Bay	America/Gran
America/Grenada	America/Guadeloupe	America/Guat
America/Guayaquil	America/Guyana	America/Halif
America/Havana	America/Hermosillo	America/India
America/Indiana/Knox	America/Indiana/Marengo	America/India
America/Indiana/Tell_City	America/Indiana/Vevay	America/India
America/Indiana/Winamac	America/Indianapolis	America/Inuvi
America/Iqaluit	America/Jamaica	America/Jujuy
America/Juneau	America/Kentucky/Louisville	America/Kent
America/Knox_IN	America/Kralendijk	America/La_F
America/Lima	America/Los_Angeles	America/Louis
America/Lower_Princes	America/Maceio	America/Mana
America/Manaus	America/Marigot	America/Marti

America/Matamoros	America/Mazatlan	America/Menc
America/Menominee	America/Merida	America/Metla
America/Mexico_City	America/Miquelon	America/Monc
America/Monterrey	America/Montevideo	America/Mont
America/Montserrat	America/Nassau	America/New_
America/Nipigon	America/Nome	America/Noro
America/North_Dakota/Beulah	America/North_Dakota/Center	America/North
America/Ojinaga	America/Panama	America/Pang
America/Paramaribo	America/Phoenix	America/Port-
America/Port_of_Spain	America/Porto_Acre	America/Porto
America/Puerto_Rico	America/Rainy_River	America/Rank
America/Recife	America/Regina	America/Reso
America/Rio_Branco	America/Rosario	America/Santa
America/Santarem	America/Santiago	America/Sante
America/Sao_Paulo	America/Scoresbysund	America/Shipr
America/Sitka	America/St_Barthelemy	America/St_Jo
America/St_Kitts	America/St_Lucia	America/St_T
America/St_Vincent	America/Swift_Current	America/Tegu
America/Thule	America/Thunder_Bay	America/Tijua
America/Toronto	America/Tortola	America/Vanc
America/Virgin	America/Whitehorse	America/Winn
America/Yakutat	America/Yellowknife	

南极洲

Antarctica/Casey	Antarctica/Davis	Antarctica/DumontDUrville	Antarctica/I
Antarctica/McMurdo	Antarctica/Palmer	Antarctica/Rothera	Antarctica/S
Antarctica/Vostok			

北冰洋

Arctic/Longyearbyen

亚洲

Asia/Aden	Asia/Almaty	Asia/Amman	Asia/Anadyr	
Asia/Aqtobe	Asia/Ashgabat	Asia/Ashkhabad	Asia/Baghdad	A
Asia/Baku	Asia/Bangkok	Asia/Beirut	Asia/Bishkek	A
Asia/Calcutta	Asia/Choibalsan	Asia/Chongqing	Asia/Chungking	A
Asia/Dacca	Asia/Damascus	Asia/Dhaka	Asia/Dili	A
Asia/Dushanbe	Asia/Gaza	Asia/Harbin	Asia/Hebron	A
Asia/Hong_Kong	Asia/Hovd	Asia/Irkutsk	Asia/Istanbul	A
Asia/Jayapura	Asia/Jerusalem	Asia/Kabul	Asia/Kamchatka	A
Asia/Kashgar	Asia/Kathmandu	Asia/Katmandu	Asia/Khandyga	A
Asia/Krasnoyarsk	Asia/Kuala_Lumpur	Asia/Kuching	Asia/Kuwait	A
Asia/Macau	Asia/Magadan	Asia/Makassar	Asia/Manila	A
Asia/Nicosia	Asia/Novokuznetsk	Asia/Novosibirsk	Asia/Omsk	A
Asia/Phnom_Penh	Asia/Pontianak	Asia/Pyongyang	Asia/Qatar	A
Asia/Rangoon	Asia/Riyadh	Asia/Saigon	Asia/Sakhalin	A
Asia/Seoul	Asia/Shanghai	Asia/Singapore	Asia/Taipei	A
Asia/Tbilisi	Asia/Tehran	Asia/Tel_Aviv	Asia/Thimbu	A
Asia/Tokyo	Asia/Ujung_Pandang	Asia/Ulaanbaatar	Asia/Ulan_Bator	A
Asia/Ust-Nera	Asia/Vientiane	Asia/Vladivostok	Asia/Yakutsk	A
Asia/Yerevan				

大西洋

Atlantic/Azores	Atlantic/Bermuda	Atlantic/Canary	Atlantic/Cape_Verde
Atlantic/Faroe	Atlantic/Jan_Mayen	Atlantic/Madeira	Atlantic/Reykjavik
Atlantic/St_Helena	Atlantic/Stanley		

大洋洲

Australia/ACT	Australia/Adelaide	Australia/Brisbane	Australia/Broke
Australia/Currie	Australia/Darwin	Australia/Eucla	Australia/Hobart
Australia/Lindeman	Australia/Lord_Howe	Australia/Melbourne	Australia/North
Australia/Perth	Australia/Queensland	Australia/South	Australia/Sydney
Australia/Victoria	Australia/West	Australia/Yancowinna	

欧洲

Europe/Amsterdam	Europe/Andorra	Europe/Athens	Europe/Belfast
Europe/Berlin	Europe/Bratislava	Europe/Brussels	Europe/Bucharest
Europe/Busingen	Europe/Chisinau	Europe/Copenhagen	Europe/Dublin
Europe/Guernsey	Europe/Helsinki	Europe/Isle_of_Man	Europe/Istanbul
Europe/Kaliningrad	Europe/Kiev	Europe/Lisbon	Europe/Ljubljana
Europe/Luxembourg	Europe/Madrid	Europe/Malta	Europe/Mariehamn
Europe/Monaco	Europe/Moscow	Europe/Nicosia	Europe/Oslo
Europe/Podgorica	Europe/Prague	Europe/Riga	Europe/Rome
Europe/San_Marino	Europe/Sarajevo	Europe/Simferopol	Europe/Skopje
Europe/Stockholm	Europe/Tallinn	Europe/Tirane	Europe/Tiraspol
Europe/Vaduz	Europe/Vatican	Europe/Vienna	Europe/Vilnius
Europe/Warsaw	Europe/Zagreb	Europe/Zaporozhye	Europe/Zurich

印度洋

Indian/Antananarivo	Indian/Chagos	Indian/Christmas	Indian/Cocos	Ind
Indian/Kerguelen	Indian/Mahe	Indian/Maldives	Indian/Mauritius	Indi
Indian/Reunion				

太平洋

Pacific/Apia	Pacific/Auckland	Pacific/Chatham	Pacific/Chuuk
Pacific/Efate	Pacific/Enderbury	Pacific/Fakaofo	Pacific/Fiji
Pacific/Galapagos	Pacific/Gambier	Pacific/Guadalcanal	Pacific/Guam
Pacific/Johnston	Pacific/Kiritimati	Pacific/Kosrae	Pacific/Kwajalein
Pacific/Marquesas	Pacific/Midway	Pacific/Nauru	Pacific/Niue
Pacific/Noumea	Pacific/Pago_Pago	Pacific/Palau	Pacific/Pitcairn
Pacific/Ponape	Pacific/Port_Moresby	Pacific/Rarotonga	Pacific/Saipan
Pacific/Tahiti	Pacific/Tarawa	Pacific/Tongatapu	Pacific/Truk
Pacific/Wallis	Pacific/Yap		

W3School Python教程

来源：

- Python教程
- Python3教程

整理：飞龙

Python 基础教程

Python 简介

Python是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。

Python的设计具有很强的可读性，相比其他语言经常使用英文关键字，其他语言的一些标点符号，它具有比其他语言更有特色语法结构。

- **Python 是一种解释型语言：** 这意味着开发过程中没有了编译这个环节。类似于PHP和Perl语言。
- **Python 是交互式语言：** 这意味着，您可以在一个Python提示符，直接互动执行写你的程序。
- **Python 是面向对象语言:** 这意味着Python支持面向对象的风格或代码封装在对象的编程技术。
- **Python是初学者的语言**：Python 对初级程序员而言，是一种伟大的语言，它支持广泛的应用程序开发，从简单的文字处理到 WWW 浏览器再到游戏。

Python发展历史

Python是由Guido van Rossum在八十年代末和九十年代初，在荷兰国家数学和计算机科学研究所设计出来的。

Python 本身也是由诸多其他语言发展而来的,这包括 ABC、Modula-3、C、C++、Algol-68、SmallTalk、Unix shell 和其他的脚本语言等等。

像Perl语言一样, Python 源代码同样遵循 GPL(GNU General Public License)协议。

现在Python是由一个核心开发团队在维护，，Guido van Rossum 仍然占据着至关重要的作用，指导其进展。

Python特点

- **1.**易于学习：Python有相对较少的关键字，结构简单，和一个明确定义的语法，学习起来更加简单。
- **2.**易于阅读：Python代码定义的更清晰。
- **3.**易于维护：Python的成功在于它的源代码是相当容易维护的。
- **4.**一个广泛的标准库：Python的最大的优势之一是丰富的库，跨平台的，在UNIX，Windows和Macintosh兼容很好。

- **5.**互动模式：互动模式的支持，您可以从终端输入并获得结果的语言，互动的测试和调试代码片断。
- **6.**便携式：Python可以运行在多种硬件平台和所有平台上都具有相同的接口。
- **7.**可扩展：可以添加低层次的模块到Python解释器。这些模块使程序员可以添加或定制自己的工具，更有效。
- **8.**数据库：Python提供所有主要的商业数据库的接口。
- **9.GUI**编程：Python支持GUI可以创建和移植到许多系统调用。
- **10.**可扩展性：相比 shell 脚本，Python 提供了一个更好的结构，且支持大型程序。

Python 环境搭建

本章节我们将向大家介绍如何在本地搭建Python开发环境。

Python可应用于多平台包括 Linux 和 Mac OS X。

你可以通过终端窗口输入 "python" 命令来查看本地是否已经安装Python以及Python的安装版本。

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, 等等。)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (多个DOS版本)
- PalmOS
- Nokia 移动手机
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python 同样可以移植到 Java 和 .NET 虚拟机上。

Python下载

Python最新源码，二进制文档，新闻资讯等可以在Python的官网查看到：

Python官网：http://www.python.org/

你可以在一下链接中下载Python的文档，你可以下载 HTML、PDF 和 PostScript 等格式的文档。

Python文档下载地址：www.python.org/doc/

Python安装

Python已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。

您需要下载适用于您使用平台的二进制代码，然后安装Python。

如果您平台的二进制代码是不可用的，你需要使用C编译器手动编译源代码。

编译的源代码，功能上有更多的选择性， 为python安装提供了更多的灵活性。

以下为不同平台上安装Python的方法：

Unix & Linux 平台安装 Python:

以下为在Unix & Linux 平台上安装 Python 的简单步骤：

- 打开WEB浏览器访问http://www.python.org/download/
- 选择使用于Unix/Linux的源码压缩包。
- 下载及解压压缩包。
- 如果你需要自定义一些选项修改*Modules/Setup*
- 执行 ./configure 脚本
- make
- make install

执行以上操作后，Python会安装在 /usr/local/bin 目录中，Python库安装在/usr/local/lib/pythonXX，XX为你使用的Python的版本号。

Window 平台安装 Python:

以下为在 Window 平台上安装 Python 的简单步骤：

- 打开WEB浏览器访问http://www.python.org/download/
- 在下载列表中选择Window平台安装包，包格式为：*python-XYZ.msi* 文件， XYZ 为你要安装的版本号。
- 要使用安装程序 *python-XYZ.msi*, Windows系统必须支持Microsoft Installer 2.0搭配使用。只要保存安装文件到本地计算机，然后运行它，看看你的机器支持MSI。Windows XP和更高版本已经有MSI，很多老机器也可以安装MSI。
- 下载后，双击下载包，进入Python安装向导，安装非常简单，你只需要使用默认的设置一直点击"下一步"直到安装完成即可。

MAC 平台安装 Python:

最近的Macs系统都自带有Python环境，但是自带的Python版本为旧版本，你可以通过链接http://www.python.org/download/mac/ 查看MAC上Python的新版功能介绍。

MAC上完整的Python安装教程你可以查看：http://www.cwi.nl/~jack/macpython.html

环境变量配置

程序和可执行文件可以在许多目录，而这些路径很可能不在操作系统提供可执行文件的搜索路径中。

path(路径)存储在环境变量中，这是由操作系统维护的一个命名的字符串。这些变量包含可用的命令行解释器和其他程序的信息。

Unix或Windows中路径变量为PATH（UNIX区分大小写，Windows不区分大小写）。

在Mac OS中，安装程序过程中改变了python的安装路径。如果你需要在其他目录引用Python，你必须在path中添加Python目录。

在 Unix/Linux 设置环境变量

- 在 **csh shell:** 输入

  ```
  setenv PATH "$PATH:/usr/local/bin/python"
  ```

  , 按下"Enter"。

- 在 **bash shell (Linux):** 输入

  ```
  export PATH="$PATH:/usr/local/bin/python"
  ```

  ，按下"Enter"。

- 在 **sh** 或者 **ksh shell:** 输入

  ```
  PATH="$PATH:/usr/local/bin/python"
  ```

  , 按下"Enter"。

注意: /usr/local/bin/python 是Python的安装目录。

在 Windows 设置环境变量

在环境变量中添加Python目录：

- 在命令提示框中**(cmd)**：输入

  ```
  path %path%;C:\Python
  ```

  , 按下"Enter"。

注意: C:\Python 是Python的安装目录。

Python 环境变量

下面几个重要的环境变量，它应用于Python：

变量名	描述
PYTHONPATH	PYTHONPATH是Python搜索路径，默认我们import的模块都会从PYTHONPATH里面寻找。
PYTHONSTARTUP	Python启动后，先寻找PYTHONSTARTUP环境变量，然后执行此文件中变量指定的执行代码。
PYTHONCASEOK	加入PYTHONCASEOK的环境变量, 就会使python导入模块的时候不区分大小写.
PYTHONHOME	另一种模块搜索路径。它通常内嵌于的PYTHONSTARTUP或PYTHONPATH目录中，使得两个模块库更容易切换。

运行Python

有三种方式可以运行Python：

1、交互式解释器：

你可以通过命令行窗口进入python并开在交互式解释器中开始编写Python代码。

你可以在Unix，DOS或任何其他提供了命令行或者shell的系统进行python编码工作。

$python # Unix/Linux

或者

python% # Unix/Linux

或者

C:>python # Windows/DOS

以下为Python命令行参数：

选项	描述
-d	在解析时显示调试信息
-O	生成优化代码 (.pyo 文件)
-S	启动时不引入查找Python路径的位置
-v	输出Python版本号
-X	从 1.6版本之后基于内建的异常（仅仅用于字符串）已过时。
-c cmd	执行 Python 脚本，并将运行结果作为 cmd 字符串。
file	在给定的python文件执行python脚本。

2、命令行脚本

在你的应用程序中通过引入解释器可以在命令行中执行Python脚本，如下所示：

$python script.py # Unix/Linux

或者

python% script.py # Unix/Linux

或者

C:>python script.py # Windows/DOS

注意：在执行脚本时，请检查脚本是否有可执行权限。

3、集成开发环境（IDE：Integrated Development Environment）

您可以使用图形用户界面（GUI）环境来编写及运行Python代码。以下推荐各个平台上使用的IDE：

- **Unix:** IDLE 是 UNIX 上最早的 Python IDE 。
- **Windows:** PythonWin 是一个 Python 集成开发环境,在许多方面都比 IDE 优秀
- **Macintosh:** Python 的 Mac 可以使用 IDLE IDE，你可以在网站上下载对应MAC的IDLE 。

继续下一章之前，请确保您的环境已搭建成功。如果你不能够建立正确的环境，那么你就可以从您的系统管理员的帮助。

在以后的章节中给出的例子已在Centos（Linux）下Python2.4.3版本测试通过。

Python 基础语法

Python语言与Perl，C和Java等语言有许多相似之处。但是，也存在一些差异。

在本章中我们将来学习Python的基础语法，让你快速学会Python编程。

第一个Python程序

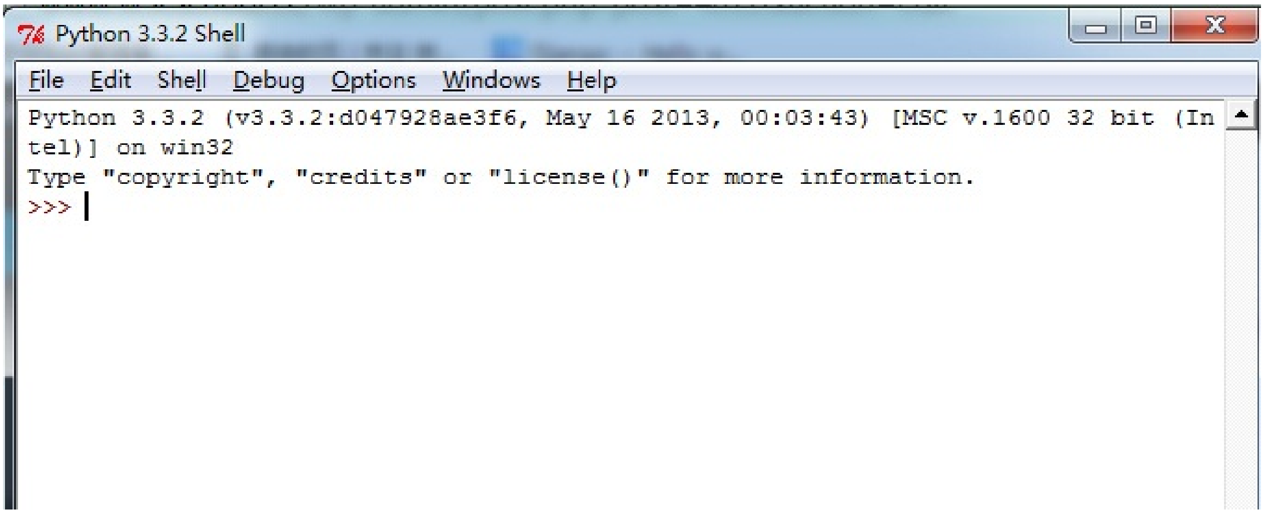
交互式编程

交互式编程不需要创建脚本文件，是通过 Python 解释器的交互模式进来编写代码。

linux上你只需要在命令行中输入 Python 命令即可启动交互式编程,提示窗口如下：

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Window上在安装Python时已经已经安装了默认的交互式编程客户端，提示窗口如下：



在 python 提示符中输入以下文本信息，然后按 Enter 键查看运行效果：

```
>>> print "Hello, Python!";
```

在 Python 2.4.3 版本中,以上事例输出结果如下：

```
Hello, Python!
```

如果您运行的是新版本的Python，那么你就需要在print语句中使用括号如：

```
>>>  print ("Hello, Python!");
```

脚本式编程

通过脚本参数调用解释器开始执行脚本，直到脚本执行完毕。当脚本执行完成后，解释器不再有效。

让我们写一个简单的Python脚本程序。所有Python文件将以.py为扩展名。将以下的源代码拷贝至test.py文件中。

```
print "Hello, Python!";
```

这里，假设你已经设置了Python解释器PATH变量。使用以下命令运行程序：

```
$ python test.py
```

输出结果：

```
Hello, Python!
```

让我们尝试另一种方式来执行Python脚本。修改test.py文件，如下所示：

```
#!/usr/bin/python

print "Hello, Python!";
```

这里，假定您的Python解释器在/usr/bin目录中，使用以下命令执行脚本：

```
$ chmod +x test.py     # 脚本文件添加可执行权限
$./test.py
```

输出结果：

```
Hello, Python!
```

Python标识符

在python里，标识符有字母、数字、下划线组成。

在python中，所有标识符可以包括英文、数字以及下划线（_），但不能以数字开头。

python中的标识符是区分大小写的。

以下划线开头的标识符是有特殊意义的。以单下划线开头（_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用"from xxx import *"而导入；

以双下划线开头的（**foo**）代表类的私有成员；以双下划线开头和结尾的（foo）代表**python**里特殊方法专用的标识，如init__（）代表类的构造函数。

Python保留字符

下面的列表显示了在Python中的保留字。这些保留字不能用作常数或变数，或任何其他标识符名称。

所有Python的关键字只包含小写字母。

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

行和缩进

学习Python与其他语言最大的区别就是，Python的代码块不使用大括号（{}）来控制类，函数以及其他逻辑判断。python最具特色的就是用缩进来写模块。

缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行。如下所示：

```
if True:
    print "True"
else:
  print "False"
```

以下代码将会执行错误：

```
 if True:
    print "Answer"
    print "True"
else:
    print "Answer"
  print "False"
```

因此，在Python的代码块中必须使用相同数目的行首缩进空格数。

以下实例包含了相同数目的行首缩进代码语句块的例子：

```
#!/usr/bin/python

import sys

try:
  # open file stream
  file = open(file_name, "w")
except IOError:
  print "There was an error writing to", file_name
  sys.exit()
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
  file_text = raw_input("Enter text: ")
  if file_text == file_finish:
    # close the file
    file.close
    break
  file.write(file_text)
  file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
  print "Next time please enter something"
  sys.exit()
try:
  file = open(file_name, "r")
except IOError:
  print "There was an error reading file"
  sys.exit()
file_text = file.read()
file.close()
print file_text
```

多行语句

Python语句中一般以新行作为为语句的结束符。

但是我们可以使用斜杠（\）将一行的语句分为多行显示，如下所示：

```
 total = item_one + \
        item_two + \
        item_three
```

语句中包含[], {} 或 () 括号就不需要使用多行连接符。如下实例：

```
days = ['Monday', 'Tuesday', 'Wednesday',
       'Thursday', 'Friday']
```

Python 引号

Python 接收单引号(')，双引号(")，三引号(''' """) 来表示字符串，引号的开始与结束必须的相同类型的。

其中三引号可以由多行组成，编写多行文本的快捷语法，常用语文档字符串，在文件的特定地点，被当做注释。

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Python注释

python中单行注释采用 # 开头。

python没有块注释，所以现在推荐的多行注释也是采用的 #比如：

```
#!/usr/bin/python

# First comment
print "Hello, Python!";  # second comment
```

输出结果：

```
Hello, Python!
```

注释可以在语句或表达式行末：

```
name = "Madisetti" # This is again comment
```

多条评论：

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Python空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行与代码缩进不同，空行并不是Python语法的一部分。书写时不插入空行，Python解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

等待用户输入

下面的程序在按回车键后就会等待用户输入：

```
#!/usr/bin/python

raw_input("\n\nPress the enter key to exit.")
```

以上代码中，"\n\n"在结果输出前会输出两个新的空行。一旦用户按下键时，程序将退出。

同一行显示多条语句

Python可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

多个语句构成代码组

缩进相同的一组语句构成一个代码块，我们称之代码组。

像if、while、def和class这样的复合语句，首行以关键字开始，以冒号(:)结束，该行之后的一行或多行代码构成代码组。

我们将首行及后面的代码组称为一个子句(clause)。

如下实例：

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

命令行参数

很多程序可以执行一些操作来查看一些基本信，Python可以使用-h参数查看各参数帮助信息：

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit

[ etc. ]
```

Python 变量类型

变量存储在内存中的值。这就意味着在创建变量时会在内存中开辟一个空间。

基于变量的数据类型，解释器会分配指定内存，并决定什么数据可以被存储在内存中。

因此，变量可以指定不同的数据类型，这些变量可以存储整数，小数或字符。

变量赋值

Python中的变量不需要声明，变量的赋值操作既是变量声明和定义的过程。

每个变量在内存中创建，都包括变量的标识，名称和数据这些信息。

每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

等号（=）用来给变量赋值。

等号（=）运算符左边是一个变量名,等号（=）运算符右边是存储在变量中的值。例如：

```
#!/usr/bin/python

counter = 100 # An integer assignment
miles = 1000.0 # A floating point
name = "John" # A string

print counter
print miles
print name
```

以上实例中，100，1000.0和"John"分别赋值给counter，miles，name变量。

执行以上程序会输出如下结果：

```
100
1000.0
John
```

多个变量赋值

Python允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

以上实例，创建一个整型对象，值为1，三个变量被分配到相同的内存空间上。

您也可以为多个对象指定多个变量。例如：

```
a, b, c = 1, 2, "john"
```

以上实例，两个整型对象1和2的分配给变量a和b，字符串对象"john"分配给变量c。

标准数据类型

在内存中存储的数据可以有多种类型。

例如，person.s年龄作为一个数值存储和他或她的地址是字母数字字符存储。

Python有一些标准类型用于定义操作上，他们和为他们每个人的存储方法可能。

Python有五个标准的数据类型：

- Numbers（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Dictionary（字典）

Python数字

数字数据类型用于存储数值。

他们是不可改变的数据类型，这意味着改变数字数据类型会分配一个新的对象。

当你指定一个值时，Number对象就会被创建：

```
var1 = 1
var2 = 10
```

您也可以使用del语句删除一些对象引用。

del语句的语法是：

```
del var1[,var2[,var3[....,varN]]]]
```

您可以通过使用del语句删除单个或多个对象。例如：

```
del var
del var_a, var_b
```

Python支持四种不同的数值类型：

- int（有符号整型）
- long（长整型[也可以代表八进制和十六进制]）
- float（浮点型）
- complex（复数）

实例

一些数值类型的实例：

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- 长整型也可以使用小写"L"，但是还是建议您使用大写"L"，避免与数字"1"混淆。Python使用"L"来显示长整型。
- Python还支持复数，复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型

Python字符串

字符串或串(String)是由数字、字母、下划线组成的一串字符。

一般记为：

```
s="a1a2···an"(n>=0)
```

它是编程语言中表示文本的数据类型。

python的字串列表有2种取值顺序:

- 从左到右索引默认0开始的，最大范围是字符串长度少1
- 从右到左索引默认-1开始的，最大范围是字符串开头

如果你的实要取得一段子串的话，可以用到变量[头下标:尾下标]，就可以截取相应的字符串，其中下标是从0开始算起，可以是正数或负数，下标可以为空表示取到头或尾。

比如:

```
s = 'ilovepython'
```

s[1:5]的结果是love。

当使用以冒号分隔的字符串，python返回一个新的对象，结果包含了以这对偏移标识的连续的内容，左边的开始是包含了下边界。

上面的结果包含了s[1]的值l，而取到的最大范围不包括上边界，就是s[5]的值p。

加号（+）是字符串连接运算符，星号（*）是重复操作。如下实例：

```
#!/usr/bin/python

str = 'Hello World!'

print str # 输出完整字符串
print str[0] # 输出字符串中的第一个字符
print str[2:5] # 输出字符串中第三个至第五个之间的字符串
print str[2:] # 输出从第三个字符开始的字符串
print str * 2 # 输出字符串两次
print str + "TEST" # 输出连接的字符串
```

以上实例输出结果：

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python列表

List（列表） 是 Python 中使用最频繁的数据类型。

列表可以完成大多数集合类的数据结构实现。它支持字符，数字，字符串甚至可以包含列表（所谓嵌套）。

列表用[ ]标识。是python最通用的复合数据类型。看这段代码就明白。

列表中的值得分割也可以用到变量[头下标:尾下标]，就可以截取相应的列表，从左到右索引默认0开始的，从右到左索引默认-1开始，下标可以为空表示取到头或尾。

加号（+）是列表连接运算符，星号（*）是重复操作。如下实例：

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list # 输出完整列表
print list[0] # 输出列表的第一个元素
print list[1:3] # 输出第二个至第三个的元素
print list[2:] # 输出从第三个开始至列表末尾的所有元素
print tinylist * 2 # 输出列表两次
print list + tinylist # 打印组合的列表
```

以上实例输出结果：

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python元组

元组是另一个数据类型，类似于List（列表）。

元组用"()"标识。内部元素用逗号隔开。但是元素不能二次赋值，相当于只读列表。

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print list # 输出完整列表
print list[0] # 输出列表的第一个元素
print list[1:3] # 输出第二个至第三个的元素
print list[2:] # 输出从第三个开始至列表末尾的所有元素
print tinylist * 2 # 输出列表两次
print list + tinylist # 打印组合的列表
```

以上实例输出结果：

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

以下是元组无效的，因为元组是不允许更新的。而列表是允许更新的：

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000 # 元组中是非法应用
list[2] = 1000 # 列表中是合法应用
```

Python元字典

字典(dictionary)是除列表意外python之中最灵活的内置数据结构类型。列表是有序的对象结合，字典是无序的对象集合。

两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

字典用"{ }"标识。字典由索引(key)和它对应的值value组成。

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one'] # 输出键为'one' 的值
print dict[2] # 输出键为 2 的值
print tinydict # 输出完整的字典
print tinydict.keys() # 输出所有键
print tinydict.values() # 输出所有值
```

输出结果为：

```
This is one This is two {'dept': 'sales', 'code': 6734, 'name': 'john'} ['dept', 'code',
```

Python数据类型转换

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

函数	描述
int(x [,base])	将x转换为一个整数
long(x [,base])	将x转换为一个长整数
float(x)	将x转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效Python表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组。
frozenset(s)	转换为不可变集合
chr(x)	将一个整数转换为一个字符
unichr(x)	将一个整数转换为Unicode字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

Python 运算符

什么是运算符？

本章节主要说明Python的运算符。举个简单的例子 **4 +5 = 9** 。 例子中，4和5被称为操作数，"+"号为运算符。

Python语言支持以下类型的运算符:

- 算术运算符
- 比较（关系）运算符
- 赋值运算符
- 逻辑运算符
- 位运算符
- 成员运算符
- 身份运算符
- 运算符优先级

接下来让我们一个个来学习Python的运算符。

Python算术运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
+	加 - 两个对象相加	a + b 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	a - b 输出结果 -10
*	乘 - 两个数相乘或是返回一个被重复若干次的字符串	a * b 输出结果 200
/	除 - x除以y	b / a 输出结果 2
%	取模 - 返回除法的余数	b % a 输出结果 0
**	幂 - 返回x的y次幂	a**b 为10的20次方， 输出结果 100000000000000000000
//	取整除 - 返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果 4.0

以下实例演示了Python所有算术运算符的操作：

```
#!/usr/bin/python

a = 21
b = 10
c = 0

c = a + b
print "Line 1 - Value of c is ", c

c = a - b
print "Line 2 - Value of c is ", c

c = a * b
print "Line 3 - Value of c is ", c

c = a / b
print "Line 4 - Value of c is ", c

c = a % b
print "Line 5 - Value of c is ", c

a = 2
b = 3
c = a**b
print "Line 6 - Value of c is ", c

a = 10
b = 5
c = a//b
print "Line 7 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 8
Line 7 - Value of c is 2
```

Python比较运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 true.
<>	不等于 - 比较两个对象是否不相等	(a <> b) 返回 true。这个运算符类似 != 。
>	大于 - 返回x是否大于y	(a > b) 返回 False。
<	小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。	(a < b) 返回 true。
>=	大于等于 - 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 - 返回x是否小于等于y。	(a <= b) 返回 true。

以下实例演示了Python所有比较运算符的操作：

```
#!/usr/bin/python

a = 21
b = 10
c = 0

if ( a == b ):
   print "Line 1 - a is equal to b"
else:
   print "Line 1 - a is not equal to b"

if ( a != b ):
   print "Line 2 - a is not equal to b"
else:
   print "Line 2 - a is equal to b"

if ( a <> b ):
   print "Line 3 - a is not equal to b"
else:
   print "Line 3 - a is equal to b"

if ( a < b ):
   print "Line 4 - a is less than b"
else:
   print "Line 4 - a is not less than b"

if ( a > b ):
   print "Line 5 - a is greater than b"
else:
   print "Line 5 - a is not greater than b"

a = 5;
b = 20;
if ( a <= b ):
   print "Line 6 - a is either less than or equal to  b"
else:
   print "Line 6 - a is neither less than nor equal to  b"

if ( b >= a ):
   print "Line 7 - b is either greater than  or equal to b"
else:
   print "Line 7 - b is neither greater than  nor equal to b"
```

以上实例输出结果：

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not equal to b
Line 4 - a is not less than b
Line 5 - a is greater than b
Line 6 - a is either less than or equal to b
Line 7 - b is either greater than or equal to b
```

Python赋值运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c = *a* 等效于 *c = c* a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a** 等效于 **c = c** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

以下实例演示了Python所有赋值运算符的操作：

```
#!/usr/bin/python

a = 21
b = 10
c = 0

c = a + b
print "Line 1 - Value of c is ", c

c += a
print "Line 2 - Value of c is ", c

c *= a
print "Line 3 - Value of c is ", c

c /= a
print "Line 4 - Value of c is ", c

c  = 2
c %= a
print "Line 5 - Value of c is ", c

c **= a
print "Line 6 - Value of c is ", c

c //= a
print "Line 7 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864
```

Python位运算符

按位运算符是把数字看作二进制来进行计算的。Python中的按位运算法则如下：

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符	(~a) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符	a >> 2 输出结果 15 ，二进制解释： 0000 1111

以下实例演示了Python所有位运算符的操作：

```
#!/usr/bin/python

a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a & b;        # 12 = 0000 1100
print "Line 1 - Value of c is ", c

c = a | b;        # 61 = 0011 1101
print "Line 2 - Value of c is ", c

c = a ^ b;        # 49 = 0011 0001
print "Line 3 - Value of c is ", c

c = ~a;           # -61 = 1100 0011
print "Line 4 - Value of c is ", c

c = a << 2;       # 240 = 1111 0000
print "Line 5 - Value of c is ", c

c = a >> 2;       # 15 = 0000 1111
print "Line 6 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Python逻辑运算符

Python语言支持逻辑运算符，以下假设变量a为10，变量b为20：

运算符	描述	实例
and	布尔"与" - 如果x为False，x and y返回False，否则它返回y的计算值。	(a and b) 返回 true。
or	布尔"或" - 如果x是True，它返回True，否则它返回y的计算值。	(a or b) 返回 true。
not	布尔"非" - 如果x为True，返回False。如果x为False，它返回True。	not(a and b) 返回 false。

以下实例演示了Python所有逻辑运算符的操作：

```
#!/usr/bin/python

a = 10
b = 20
c = 0

if ( a and b ):
   print "Line 1 - a and b are true"
else:
   print "Line 1 - Either a is not true or b is not true"

if ( a or b ):
   print "Line 2 - Either a is true or b is true or both are true"
else:
   print "Line 2 - Neither a is true nor b is true"

a = 0
if ( a and b ):
   print "Line 3 - a and b are true"
else:
   print "Line 3 - Either a is not true or b is not true"

if ( a or b ):
   print "Line 4 - Either a is true or b is true or both are true"
else:
   print "Line 4 - Neither a is true nor b is true"

if not( a and b ):
   print "Line 5 - a and b are true"
else:
   print "Line 5 - Either a is not true or b is not true"
```

以上实例输出结果：

```
Line 1 - a and b are true
Line 2 - Either a is true or b is true or both are true
Line 3 - Either a is not true or b is not true
Line 4 - Either a is true or b is true or both are true
Line 5 - a and b are true
```

Python成员运算符

除了以上的一些运算符之外，Python还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符	描述	实例
in	如果在指定的序列中找到值返回True，否则返回False。	x 在 y序列中，如果x在y序列中返回True。
not in	如果在指定的序列中没有找到值返回True，否则返回False。	x 不在 y序列中，如果x不在y序列中返回True。

以下实例演示了Python所有成员运算符的操作：

```
#!/usr/bin/python

a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
   print "Line 1 - a is available in the given list"
else:
   print "Line 1 - a is not available in the given list"

if ( b not in list ):
   print "Line 2 - b is not available in the given list"
else:
   print "Line 2 - b is available in the given list"

a = 2
if ( a in list ):
   print "Line 3 - a is available in the given list"
else:
   print "Line 3 - a is not available in the given list"
```

以上实例输出结果：

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Python身份运算符

身份运算符用于比较两个对象的存储单元

运算符	描述	实例
is	is是判断两个标识符是不是引用自一个对象	x is y, 如果 id(x) 等于 id(y) , **is** 返回结果 1
is not	is not是判断两个标识符是不是引用自不同对象	x is not y, 如果 id(x) 不等于 id(y). **is not** 返回结果 1

以下实例演示了Python所有身份运算符的操作：

```
#!/usr/bin/python

a = 20
b = 20

if ( a is b ):
   print "Line 1 - a and b have same identity"
else:
   print "Line 1 - a and b do not have same identity"

if ( id(a) == id(b) ):
   print "Line 2 - a and b have same identity"
else:
   print "Line 2 - a and b do not have same identity"

b = 30
if ( a is b ):
   print "Line 3 - a and b have same identity"
else:
   print "Line 3 - a and b do not have same identity"

if ( a is not b ):
   print "Line 4 - a and b do not have same identity"
else:
   print "Line 4 - a and b have same identity"
```

以上实例输出结果：

```
Line 1 - a and b have same identity
Line 2 - a and b have same identity
Line 3 - a and b do not have same identity
Line 4 - a and b do not have same identity
```

Python运算符优先级

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符
&	位 'AND'
^ |	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += = *=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

以下实例演示了Python所有运算符优先级的操作：

```
#!/usr/bin/python

a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d       #( 30 * 15 ) / 5
print "Value of (a + b) * c / d is ",  e

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print "Value of ((a + b) * c) / d is ",  e

e = (a + b) * (c / d);    # (30) * (15/5)
print "Value of (a + b) * (c / d) is ",  e

e = a + (b * c) / d;      #  20 + (150/5)
print "Value of a + (b * c) / d is ",  e
```

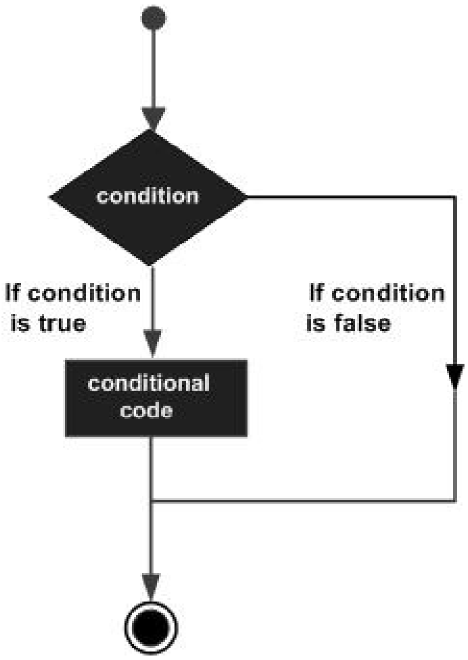
以上实例输出结果：

```
Value of (a + b) * c / d is 90
Value of ((a + b) * c) / d is 90
Value of (a + b) * (c / d) is 90
Value of a + (b * c) / d is 50
```

Python 条件语句

Python条件语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程:



Python程序语言指定任何非0和非空（null）值为true，0 或者 null为false。

Python 编程中 if 语句用于控制程序的执行，基本形式为：

```
if 判断条件：
    执行语句……
else：
    执行语句……
```

其中"判断条件"成立时（非零），则执行后面的语句，而执行内容可以多行，以缩进来区分表示同一范围。

else 为可选语句，当需要在条件不成立时执行内容则可以执行相关语句，具体例子如下：

```
# coding=utf8
# 例1：if 基本用法

flag = False
name = 'luren'
if name == 'python':         # 判断变量否为'python'
    flag = True          # 条件成立时设置标志为真
    print 'welcome boss'    # 并输出欢迎信息
else:
    print name              # 条件不成立时输出变量名称
```

输出结果为：

```
>>> luren            # 输出结果
```

if 语句的判断条件可以用>（大于）、<(小于)、==（等于）、>=（大于等于）、<=（小于等于）来表示其关系。

当判断条件为多个值是，可以使用以下形式：

```
if 判断条件1:
    执行语句1……
elif 判断条件2:
    执行语句2……
elif 判断条件3:
    执行语句3……
else:
    执行语句4……
```

实例如下：

```
# coding=utf8
# 例2：elif用法

num = 5
if num == 3:            # 判断num的值
    print 'boss'
elif num == 2:
    print 'user'
elif num == 1:
    print 'worker'
elif num < 0:           # 值小于零时输出
    print 'error'
else:
    print 'roadman'     # 条件均不成立时输出
```

输出结果为：

```
>>> roadman         # 输出结果
```

由于 python 并不支持 switch 语句，所以多个条件判断，只能用 elif 来实现，如果判断需要多个条件需同时判断时，可以使用 or （或），表示两个条件有一个成立时判断条件成功；使用 and （与）时，表示只有两个条件同时成立的情况下，判断条件才成功。

```
# coding=utf8
# 例3：if语句多个条件

num = 9
if num >= 0 and num <= 10:    # 判断值是否在0~10之间
    print 'hello'
>>> hello       # 输出结果

num = 10
if num < 0 or num > 10:    # 判断值是否在小于0或大于10
    print 'hello'
else:
    print 'undefine'
>>> undefine       # 输出结果

num = 8
# 判断值是否在0~5或者10~15之间
if (num >= 0 and num <= 5) or (num >= 10 and num <= 15):
    print 'hello'
else:
    print 'undefine'
>>> undefine       # 输出结果
```

当if有多个条件时可使用括号来区分判断的先后顺序，括号中的判断优先执行，此外 and 和 or 的优先级低于>（大于）、<（小于）等判断符号，即大于和小于在没有括号的情况下会比与或要优先判断。

简单的语句组

你也可以在同一行的位置上使用if条件判断语句，如下实例：

```
#!/usr/bin/python

var = 100

if ( var  == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

以上代码执行输出结果如下：

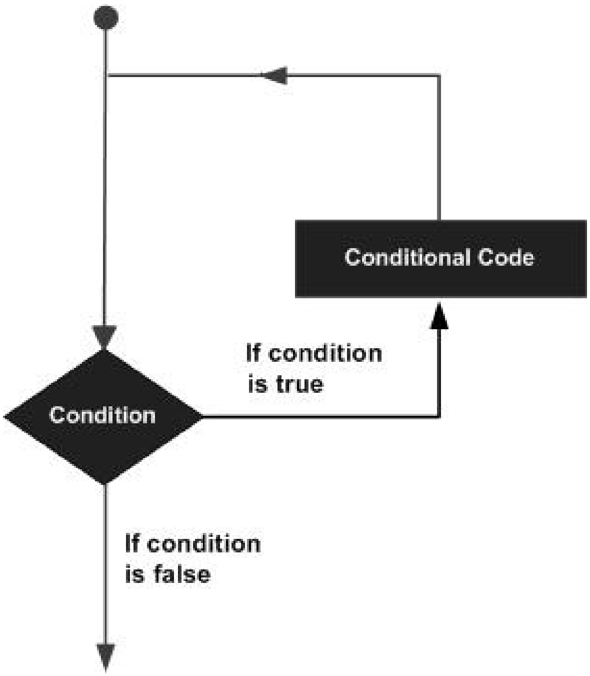
```
Value of expression is 100
Good bye!
```

Python 循环语句

本章节将向大家介绍Python的循环语句，程序在一般情况下是按顺序执行的。

编程语言提供了各种控制结构，允许更复杂的执行路径。

循环语句允许我们执行一个语句或语句组多次，下面是在大多数编程语言中的循环语句的一般形式：



Python提供了for循环和while循环（在Python中没有do..while循环）：

循环类型	描述
while 循环	在给定的判断条件为 true 时执行循环体，否则退出循环体。
for 循环	重复执行语句
嵌套循环	你可以在while循环体中嵌套for循环

循环控制语句

循环控制语句可以更改语句执行的顺序。Python支持以下循环控制语句：

控制语句	描述
break 语句	在语句块执行过程中终止循环，并且跳出整个循环
continue 语句	在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环。
pass 语句	pass是空语句，是为了保持程序结构的完整性。

Python While循环语句

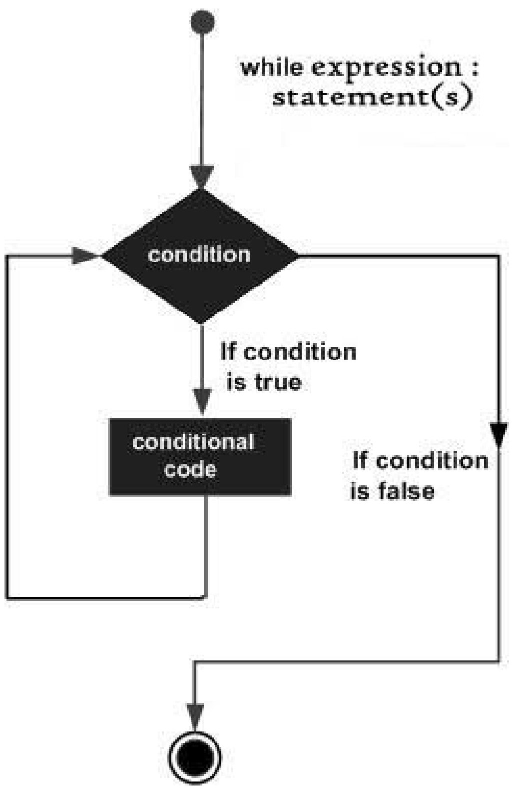
Python 编程中 while 语句用于循环执行程序，即在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。其基本形式为：

```
while 判断条件：
    执行语句……
```

执行语句可以是单个语句或语句块。判断条件可以是任何表达式，任何非零、或非空（null）的值均为true。

当判断条件假false时，循环结束。

执行流程图如下：



实例：

```
#!/usr/bin/python

count = 0
while (count < 9):
   print 'The count is:', count
   count = count + 1

print "Good bye!"
```

以上代码执行输出结果:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

while 语句时还有另外两个重要的命令 continue，break 来跳过循环，continue 用于跳过该次循环，break 则是用于退出循环，此外"判断条件"还可以是个常值，表示循环必定成立，具体用法如下：

```
# continue 和 break 用法

i = 1
while i < 10:
    i += 1
    if i%2 > 0:     # 非双数时跳过输出
        continue
    print i         # 输出双数2、4、6、8、10

i = 1
while 1:            # 循环条件为1必定成立
    print i         # 输出1~10
    i += 1
    if i > 10:     # 当i大于10时跳出循环
        break
```

无限循环

如果条件判断语句永远为 true，循环将会无限的执行下去，如下实例：

```
#!/usr/bin/python

var = 1
while var == 1 :  # 该条件永远为true，循环将无限执行下去
   num = raw_input("Enter a number  :")
   print "You entered: ", num

print "Good bye!"
```

以上实例输出结果：

```
Enter a number  :20
You entered:  20
Enter a number  :29
You entered:  29
Enter a number  :3
You entered:  3
Enter a number between :Traceback (most recent call last):
  File "test.py", line 5, in <module>
    num = raw_input("Enter a number :")
KeyboardInterrupt
```

注意：以上的无限循环你可以使用 CTRL+C 来中断循环。

循环使用 else 语句

在 python 中，for … else 表示这样的意思，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

```
#!/usr/bin/python

count = 0
while count < 5:
   print count, " is  less than 5"
   count = count + 1
else:
   print count, " is not less than 5"
```

以上实例输出结果为：

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

简单语句组

类似if语句的语法，如果你的while循环体中只有一条语句，你可以将该语句与while写在同一行中， 如下所示：

```
#!/usr/bin/python

flag = 1

while (flag): print 'Given flag is really true!'

print "Good bye!"
```

注意：以上的无限循环你可以使用 CTRL+C 来中断循环。

Python for 循环语句

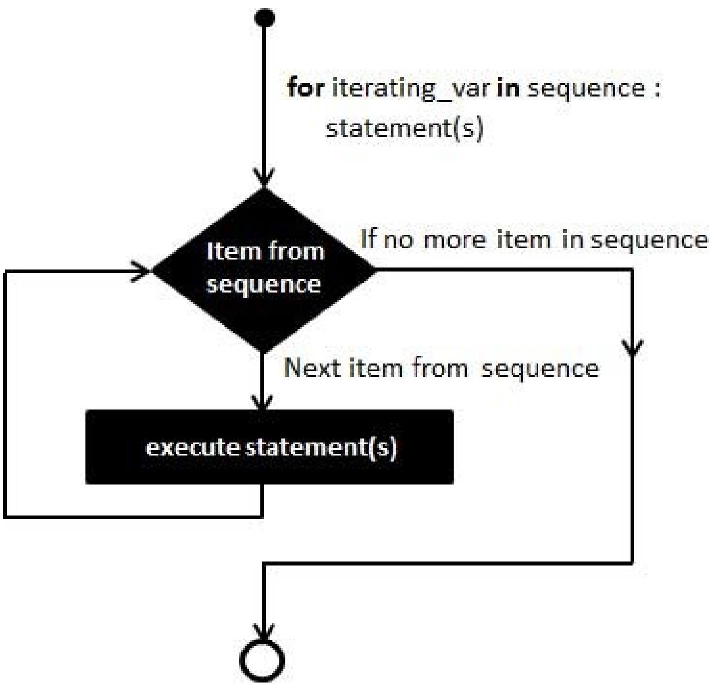
Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

语法：

for循环的语法格式如下：

```
for iterating_var in sequence:
   statements(s)
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print 'Current fruit :', fruit

print "Good bye!"
```

以上实例输出结果:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

通过序列索引迭代

另外一种执行循环的遍历方式是通过索引，如下实例：

```
#!/usr/bin/python

fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]

print "Good bye!"
```

以上实例输出结果：

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

以上实例我们使用了内置函数 len() 和 range(),函数 len() 返回列表的长度，即元素的个数。range返回一个序列的数。

循环使用 else 语句

在 python 中，for … else 表示这样的意思，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

如下实例：

```
#!/usr/bin/python

for num in range(10,20):  #to iterate between 10 to 20
   for i in range(2,num): #to iterate on the factors of the number
      if num%i == 0:      #to determine the first factor
         j=num/i          #to calculate the second factor
         print '%d equals %d * %d' % (num,i,j)
         break #to move to the next number, the #first FOR
   else:                  # else part of the loop
      print num, 'is a prime number'
```

以上实例输出结果：

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Python 循环嵌套

Python 语言允许在一个循环体里面嵌入另一个循环。

Python for 循环嵌套语法：

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

Python while 循环嵌套语法：

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

你可以在循环体内嵌入其他的循环体，如在while循环中可以嵌入for循环， 反之，你可以在for循环中嵌入while循环。

实例：

以下实例使用了嵌套循环输出2~100之间的素数：

```
#!/usr/bin/python

i = 2
while(i < 100):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j) : print i, " 是素数"
   i = i + 1

print "Good bye!"
```

以上实例输出结果:

```
2 是素数
3 是素数
5 是素数
7 是素数
11 是素数
13 是素数
17 是素数
19 是素数
23 是素数
29 是素数
31 是素数
37 是素数
41 是素数
43 是素数
47 是素数
53 是素数
59 是素数
61 是素数
67 是素数
71 是素数
73 是素数
79 是素数
83 是素数
89 是素数
97 是素数
Good bye!
```

Python break 语句

Python break语句，就像在C语言中，打破了最小封闭for或while循环。

break语句用来终止循环语句，即循环条件没有False条件或者序列还没被完全递归完，也会停止执行循环语句。

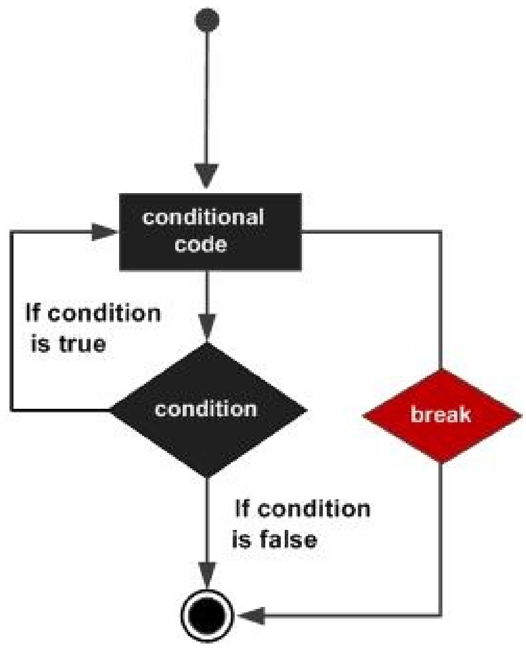
break语句用在while和for循环中。

如果您使用嵌套循环，break语句将停止执行最深层的循环，并开始执行下一行代码。

Python语言 break 语句语法：

```
break
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

Python continue 语句

Python continue 语句跳出本次循环，而break跳出整个循环。

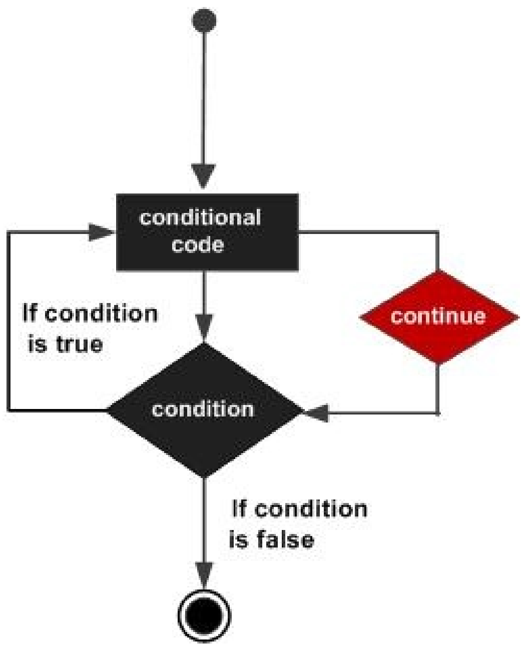
continue 语句用来告诉Python跳过当前循环的剩余语句，然后继续进行下一轮循环。

continue语句用在while和for循环中。

Python 语言 continue 语句语法格式如下：

```
continue
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print 'Current variable value :', var
print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Python pass 语句

Python pass是空语句，是为了保持程序结构的完整性。

Python 语言 pass 语句语法格式如下：

```
pass
```

实例：

```
#!/usr/bin/python

for letter in 'Python':
   if letter == 'h':
      pass
      print 'This is pass block'
   print 'Current Letter :', letter

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Python 数字

Python 数字数据类型用于存储数值。

数据类型是不允许改变的,这就意味着如果改变数字数据类型得值，将重新分配内存空间。

以下实例在变量赋值时数字对象将被创建：

```
var1 = 1
var2 = 10
```

您也可以使用del语句删除一些数字对象引用。

del语句的语法是：

```
del var1[,var2[,var3[....,varN]]]]
```

您可以通过使用del语句删除单个或多个对象，例如：

```
del var
del var_a, var_b
```

Python 支持四种不同的数值类型：

- 整型**(Int)** - 通常被称为是整型或整数，是正或负整数，不带小数点。
- 长整型**(long integers)** - 无限大小的整数，整数最后是一个大写或小写的L。
- 浮点型**(floating point real values)** - 浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示（2.5e2 = 2.5 x 10^2 = 250）
- 复数**((complex numbers))** - 复数的虚部以字母J 或 j结尾 。如：2+3i

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- 长整型也可以使用小写"L"，但是还是建议您使用大写"L"，避免与数字"1"混淆。Python使

用"L"来显示长整型。

- Python还支持复数，复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型

Python数字类型转换

```
int(x [,base ])         将x转换为一个整数
long(x [,base ])        将x转换为一个长整数
float(x )               将x转换到一个浮点数
complex(real [,imag ])  创建一个复数
str(x )                 将对象 x 转换为字符串
repr(x )                将对象 x 转换为表达式字符串
eval(str )              用来计算在字符串中的有效Python表达式,并返回一个对象
tuple(s )               将序列 s 转换为一个元组
list(s )                将序列 s 转换为一个列表
chr(x )                 将一个整数转换为一个字符
unichr(x )              将一个整数转换为Unicode字符
ord(x )                 将一个字符转换为它的整数值
hex(x )                 将一个整数转换为一个十六进制字符串
oct(x )                 将一个整数转换为一个八进制字符串
```

Python数学函数

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如math.ceil(4.1) 返回 5
cmp(x, y)	如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1
exp(x)	返回e的x次幂(e^x),如math.exp(1) 返回2.718281828459045
fabs(x)	返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)	返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)	如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)	返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
round(x [,n])	返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
sqrt(x)	返回数字x的平方根，数字可以为负数，返回类型为实数，如math.sqrt(4)返回 2+0j

Python随机数函数

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。

Python包含以下常用随机数函数：

函数	描述
choice(seq)	从序列的元素中随机挑选一个元素，比如random.choice(range(10))，从0到9中随机挑选一个整数。
randrange ([start,] stop [,step])	从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺省值为1
random()	随机生成下一个实数，它在[0,1)范围内。
seed([x])	改变随机数生成器的种子seed。如果你不了解其原理，你不必特别去设定seed，Python会帮你选择seed。
shuffle(lst)	将序列的所有元素随机排序
uniform(x, y)	随机生成下一个实数，它在[x,y]范围内。

Python三角函数

Python包括以下三角函数：

函数	描述
acos(x)	返回x的反余弦弧度值。
asin(x)	返回x的反正弦弧度值。
atan(x)	返回x的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回x的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(*xx* + *yy*)。
sin(x)	返回的x弧度的正弦值。
tan(x)	返回x弧度的正切值。
degrees(x)	将弧度转换为角度,如degrees(math.pi/2)， 返回90.0
radians(x)	将角度转换为弧度

Python数学常量

常量	描述
pi	数学常量 pi（圆周率，一般以π来表示）
e	数学常量 e，e即自然常数（自然常数）。

Python 字符串

字符串是最 Python 总常用的数据类型。我们可以使用引号来创建字符串。

创建字符串很简单，只要为变量分配一个值即可。例如：

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Python访问字符串中的值

Python不支持单字符类型，单字符也在Python也是作为一个字符串使用。

Python访问子字符串，可以使用方括号来截取字符串，如下实例：

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

以上实例执行结果：

```
var1[0]:  H
var2[1:5]:  ytho
```

Python字符串更新

你可以对已存在的字符串进行修改，并赋值给另一个变量，如下实例：

```
#!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'
```

以上实例执行结果

```
Updated String :-  Hello Python
```

Python转义字符

在需要在字符中使用特殊字符时，python用反斜杠()转义字符。如下表：

转义字符	描述
(在行尾时)	续行符
\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy代表的字符，例如：\o12代表换行
\xyy	十六进制数，yy代表的字符，例如：\x0a代表换行
\other	其它的字符以普通格式输出

Python字符串运算符

下表实例变量a值为字符串"Hello"，b变量值为"Python"：

操作符	描述	实例
+	字符串连接	a + b 输出结果：HelloPython
*	重复输出字符串	a*2 输出结果：HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 **e**
[:]	截取字符串中的一部分	a[1:4] 输出结果 **ell**
in	成员运算符 - 如果字符串中包含给定的字符返回 True	**H in a** 输出结果 1
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	**M not in a** 输出结果 1
r/R	原始字符串 - 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。 原始字符串除在字符串的第一个引号前加上字母"r"（可以大小写）以外，与普通字符串有着几乎完全相同的语法。	**print r'\n'** prints \n 和 **print R'\n'** prints \n
%	格式字符串	情看一下章节

Python字符串格式化

Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。

在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。

如下实例：

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

以上实例输出结果：

```
My name is Zara and weight is 21 kg!
```

python字符串格式化符号:

符 号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

格式化操作符辅助指令:

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号(+)
<sp>	在正数前面显示空格
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

Python三引号（triple quotes）

python中三引号可以将复杂的字符串进行复制:

python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。

三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

```
 >>> hi = '''hi
there'''
>>> hi   # repr()
'hi\nthere'
>>> print hi  # str()
hi
there
```

三引号让程序员从引号和特殊字符串的泥潭里面解脱出来，自始至终保持一小块字符串的格式是所谓的WYSIWYG（所见即所得）格式的。

一个典型的用例是，当你需要一块HTML或者SQL时，这时用字符串组合，特殊字符串转义将会非常的繁琐。

```
 errHTML = '''
<HTML><HEAD><TITLE>
Friends CGI Demo</TITLE></HEAD>
<BODY><H3>ERROR</H3>
<B>%s</B><P>
<FORM><INPUT TYPE=button VALUE=Back
ONCLICK="window.history.back()"></FORM>
</BODY></HTML>
'''
cursor.execute('''
CREATE TABLE users (
login VARCHAR(8),
uid INTEGER,
prid INTEGER)
''')
```

Unicode 字符串

Python 中定义一个 Unicode 字符串和定义一个普通字符串一样简单：

```
>>> u'Hello World !'
u'Hello World !'
```

引号前小写的"u"表示这里创建的是一个 Unicode 字符串。如果你想加入一个特殊字符，可以使用 Python 的 Unicode-Escape 编码。如下例所示：

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

被替换的 \u0020 标识表示在给定位置插入编码值为 0x0020 的 Unicode 字符（空格符）。

python的字符串内建函数

字符串方法是从python1.6到2.0慢慢加进来的——它们也被加到了Jython中。

这些方法实现了string模块的大部分方法，如下表所示列出了目前字符串内建支持的方法，所有的方法都包含了对Unicode的支持，有一些甚至是专门用于Unicode的。

方法	描述
string.capitalize()	把字符串的第一个字符大写
string.center(width)	返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
string.count(str, beg=0, end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
string.decode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式解码 string，如果出错默认报一个 ValueError 的 异 常 ， 除 非 errors 指 定 的 是 'ignore' 或 者'replace'
string.encode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式编码 string，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string.endswith(obj, beg=0, end=len(string))	检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.
string.expandtabs(tabsize=8)	把字符串 string 中的 tab 符号转为空格，默认的空格数 tabsize 是 8.
string.find(str, beg=0, end=len(string))	检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1
string.index(str, beg=0, end=len(string))	跟find()方法一样，只不过如果str不在 string中会报一个异常.
string.isalnum()	如果 string 至少有一个字符并且所有字符都是字母或数字则返 回 True,否则返回 False
string.isalpha()	如果 string 至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False
string.isdecimal()	如果 string 只包含十进制数字则返回 True 否则返回 False.
string.isdigit()	如果 string 只包含数字则返回 True 否则返回 False.
string.islower()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
string.isnumeric()	如果 string 中只包含数字字符，则返回 True，否则返回 False
string.isspace()	如果 string 中只包含空格，则返回 True，否则返回 False.

string.istitle()	如果 string 是标题化的(见 title())则返回 True，否则返回 False
string.isupper()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
string.join(seq)	Merges (concatenates)以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string.ljust(width)	返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串
string.lower()	转换 string 中所有大写字符为小写.
string.lstrip()	截掉 string 左边的空格
string.maketrans(intab, outtab])	maketrans() 方法用于创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
max(str)	返回字符串 *str* 中最大的字母。
min(str)	返回字符串 *str* 中最小的字母。
string.partition(str)	有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把 字 符 串 string 分 成 一 个 3 元 素 的 元 组 (string_pre_str,str,string_post_str),如果 string 中不包含str 则 string_pre_str == string.
string.replace(str1, str2, num=string.count(str1))	把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.
string.rfind(str, beg=0,end=len(string))	类似于 find()函数，不过是从右边开始查找.
string.rindex(str, beg=0,end=len(string))	类似于 index()，不过是从右边开始.
string.rjust(width)	返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
string.rpartition(str)	类似于 partition()函数,不过是从右边开始查找.
string.rstrip()	删除 string 字符串末尾的空格.
string.split(str="", num=string.count(str))	以 str 为分隔符切片 string，如果 num有指定值，则仅分隔 num 个子字符串
string.splitlines(num=string.count('\n'))	按照行分隔，返回一个包含各行作为元素的列表，如果 num 指定则仅切片 num 个行.
string.startswith(obj, beg=0,end=len(string))	检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查.

string.strip([obj])	在 string 上执行 lstrip()和 rstrip()
string.swapcase()	翻转 string 中的大小写
string.title()	返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
string.translate(str, del="")	根据 str 给出的表(包含 256 个字符)转换 string 的字符, 要过滤掉的字符放到 del 参数中
string.upper()	转换 string 中的小写字母为大写
string.zfill(width)	返回长度为 width 的字符串，原字符串 string 右对齐，前面填充0
string.isdecimal()	isdecimal()方法检查字符串是否只包含十进制字符。这种方法只存在于unicode对象。

Python 列表(Lists)

序列是Python中最基本的数据结构。序列中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是0，第二个索引是1，依此类推。

Python有6个序列的内置类型，但最常见的是列表和元组。

序列都可以进行的操作包括索引，切片，加，乘，检查成员。

此外，Python已经内置确定序列的长度以及确定最大和最小的元素的方法。

列表是最常用的Python数据类型，它可以作为一个方括号内的逗号分隔值出现。

列表的数据项不需要具有相同的类型

创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。如下所示：

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

与字符串的索引一样，列表索引从0开始。列表可以进行截取、组合等。

访问列表中的值

使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符，如下所示：

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];

print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

以上实例输出结果：

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

更新列表

你可以对列表的数据项进行修改或更新，你也可以使用append()方法来添加列表项，如下所示：

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "
print list[2];
list[2] = 2001;
print "New value available at index 2 : "
print list[2];
```

注意：我们会在接下来的章节讨论append()方法的使用

以上实例输出结果：

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

删除列表元素

可以使用 del 语句来删除列表的的元素，如下实例：

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];

print list1;
del list1[2];
print "After deleting value at index 2 : "
print list1;
```

以上实例输出结果：

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

注意：我们会在接下来的章节讨论remove()方法的使用

Python列表脚本操作符

列表对 + 和 的操作符与字符串相似。+ 号用于组合列表， 号用于重复列表。

如下所示：

Python 表达式	结果	描述
len([1, 2, 3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print x,	1 2 3	迭代

Python列表截取

Python的列表截取与字符串操作类型，如下所示：

```
L = ['spam', 'Spam', 'SPAM!']
```

操作：

Python 表达式	结果	描述
L[2]	'SPAM!'	读取列表中第三个元素
L[-2]	'Spam'	读取列表中倒数第二个元素
L[1:]	['Spam', 'SPAM!']	从第二个元素开始截取列表

Python列表函数&方法

Python包含以下函数:

函数	描述
cmp(list1, list2)	比较两个列表的元素
len(list)	列表元素个数
max(list)	返回列表元素最大值
min(list)	返回列表元素最小值
list(seq)	将元组转换为列表

Python包含以下方法:

方法	描述
list.append(obj)	在列表末尾添加新的对象
list.count(obj)	统计某个元素在列表中出现的次数
list.extend(seq)	在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
list.index(obj)	从列表中找出某个值第一个匹配项的索引位置
list.insert(index, obj)	将对象插入列表
list.pop(obj=list[-1])	移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
list.remove(obj)	移除列表中某个值的第一个匹配项
list.reverse()	反向列表中元素
list.sort([func])	对原列表进行排序

Python 元组

Python的元组与列表类似，不同之处在于元组的元素不能修改。

元组使用小括号，列表使用方括号。

元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

如下实例：

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

创建空元组

```
tup1 = ();
```

元组中只包含一个元素时，需要在元素后面添加逗号

```
tup1 = (50,);
```

元组与字符串类似，下标索引从0开始，可以进行截取，组合等。

访问元组

元组可以使用下标索引来访问元组中的值，如下实例:

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
```

以上实例输出结果：

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例:

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# 以下修改元组元素操作是非法的。
# tup1[0] = 100;

# 创建一个新的元组
tup3 = tup1 + tup2;
print tup3;
```

以上实例输出结果：

```
(12, 34.56, 'abc', 'xyz')
```

删除元组

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组，如下实例:

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup;
del tup;
print "After deleting tup : "
print tup;
```

以上实例元组被删除后，输出变量会有异常信息，输出如下所示：

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

元组运算符

与字符串一样，元组之间可以使用 + 号和 * 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

Python 表达式	结果	描述
len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
['Hi!'] * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
3 in (1, 2, 3)	True	元素是否存在
for x in (1, 2, 3): print x,	1 2 3	迭代

元组索引，截取

因为元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素，如下所示：

元组：

```
L = ('spam', 'Spam', 'SPAM!')
```

Python 表达式	结果	描述
L[2]	'SPAM!'	读取第三个元素
L[-2]	'Spam'	反向读取；读取倒数第二个元素
L[1:]	['Spam', 'SPAM!']	截取元素

无关闭分隔符

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

以上实例允许结果：

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

元组内置函数

Python元组包含了以下内置函数

方法	描述
cmp(tuple1, tuple2)	比较两个元组元素。
len(tuple)	计算元组元素个数。
max(tuple)	返回元组中元素最大值。
min(tuple)	返回元组中元素最小值。
tuple(seq)	将列表转换为元组。

Python 字典(Dictionary)

字典是另一种可变容器模型，且可存储任意类型对象，如其他容器模型。

字典由键和对应值成对组成。字典也被称作关联数组或哈希表。基本语法如下：

```
dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

也可如此创建字典：

```
dict1 = { 'abc': 456 };
dict2 = { 'abc': 123, 98.6: 37 };
```

每个键与值用冒号隔开（:），每对用逗号，每对用逗号分割，整体放在花括号中（{}）。

键必须独一无二，但值则不必。

值可以取任何数据类型，但必须是不可变的，如字符串，数或元组。

访问字典里的值

把相应的键放入熟悉的方括弧，如下实例:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Name']: ", dict['Name'];
print "dict['Age']: ", dict['Age'];
```

以上实例输出结果：

```
dict['Name']:  Zara
dict['Age']:  7
```

如果用字典里没有的键访问数据，会输出错误如下：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Alice']: ", dict['Alice'];
```

以上实例输出结果：

```
dict['Zara']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age'];
print "dict['School']: ", dict['School'];
```

以上实例输出结果：

```
dict['Age']:  8
dict['School']:  DPS School
```

删除字典元素

能删单一的元素也能清空字典，清空只需一项操作。

显示删除一个字典用del命令，如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
del dict ;        # 删除词典

print "dict['Age']: ", dict['Age'];
print "dict['School']: ", dict['School'];
```

但这会引发一个异常，因为用del后字典不再存在：

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

注：del()方法后面也会讨论。

删除字典元素

字典键的特性

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

两个重要的点需要记住：

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

print "dict['Name']: ", dict['Name'];
```

以上实例输出结果：

```
dict['Name']:  Manni
```

2）键必须不可变，所以可以用数，字符串或元组充当，所以用列表就不行，如下实例：

```
#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7};

print "dict['Name']: ", dict['Name'];
```

以上实例输出结果：

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = {['Name']: 'Zara', 'Age': 7};
TypeError: list objects are unhashable
```

字典内置函数&方法

Python字典包含了以下内置函数：

函数	描述
cmp(dict1, dict2)	比较两个字典元素。
len(dict)	计算字典元素个数，即键的总数。
str(dict)	输出字典可打印的字符串表示。
type(variable)	返回输入的变量类型，如果变量是字典就返回字典类型。

Python字典包含了以下内置函数：

序号	函数及描述
radiansdict.clear()	删除字典内所有元素
radiansdict.copy()	返回一个字典的浅复制
radiansdict.fromkeys()	创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值
radiansdict.get(key, default=None)	返回指定键的值，如果值不在字典中返回default值
radiansdict.has_key(key)	如果键在字典dict里返回true，否则返回false
radiansdict.items()	以列表返回可遍历的(键, 值) 元组数组
radiansdict.keys()	以列表返回一个字典所有的键
radiansdict.setdefault(key, default=None)	和get()类似, 但如果键不已经存在于字典中，将会添加键并将值设为default
radiansdict.update(dict2)	把字典dict2的键/值对更新到dict里
radiansdict.values()	以列表返回字典中的所有值

Python 日期和时间

Python程序能用很多方式处理日期和时间。转换日期格式是一个常见的例行琐事。Python有一个time and calendar模组可以帮忙。

什么是Tick？

时间间隔是以秒为单位的浮点小数。

每个时间戳都以自从1970年1月1日午夜（历元）经过了多长时间来表示。

Python附带的受欢迎的time模块下有很多函数可以转换常见日期格式。如函数time.time()用ticks计时单位返回从12:00am, January 1, 1970(epoch) 开始的记录的当前操作系统时间, 如下实例:

```
#!/usr/bin/python
import time;  # This is required to include time module.

ticks = time.time()
print "Number of ticks since 12:00am, January 1, 1970:", ticks
```

以上实例输出结果：

```
Number of ticks since 12:00am, January 1, 1970: 7186862.73399
```

Tick单位最适于做日期运算。但是1970年之前的日期就无法以此表示了。太遥远的日期也不行，UNIX和Windows只支持到2038年某日。

什么是时间元组？

很多Python函数用一个元组装起来的9组数字处理时间:

字段	值
4位数年	2008
月	1 到 12
日	1到31
小时	0到23
分钟	
秒	0到61 (60或61 是闰秒)
一周的第几日	0到6 (0是周一)
一年的第几日	1到366 (儒略历)
夏令时	-1, 0, 1, -1是决定是否为夏令时的旗帜

上述也就是struct_time元组。这种结构具有如下属性：

属性	值
tm_year	2008
tm_mon	1 到 12
tm_mday	1 到 31
tm_hour	0 到 23
tm_min	0 到 59
tm_sec	0 到 61 (60或61 是闰秒)
tm_wday	0到6 (0是周一)
tm_yday	1 到 366(儒略历)
tm_isdst	-1, 0, 1, -1是决定是否为夏令时的旗帜

获取当前时间

从返回浮点数的时间辍方式向时间元组转换，只要将浮点数传递给如localtime之类的函数。

```
#!/usr/bin/python
import time;

localtime = time.localtime(time.time())
print "Local current time :", localtime
```

以上实例输出结果：

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,
tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198, tm_isdst=0)
```

获取格式化的时间

你可以根据需求选取各种格式，但是最简单的获取可读的时间模式的函数是asctime():

```
#!/usr/bin/python
import time;

localtime = time.asctime( time.localtime(time.time()) )
print "Local current time :", localtime
```

以上实例输出结果：

```
Local current time : Tue Jan 13 10:17:09 2009
```

获取某月日历

Calendar模块有很广泛的方法用来处理年历和月历，例如打印某月的月历：

```
#!/usr/bin/python
import calendar

cal = calendar.month(2008, 1)
print "Here is the calendar:"
print cal;
```

以上实例输出结果：

```
Here is the calendar:
    January 2008
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Time模块

Time模块包含了以下内置函数，既有时间处理相的，也有转换时间格式的：

函数	描述
time.altzone	返回格林威治西部的夏令时地区的偏移秒数。如果该地区在格林威治东部会返回负值（如西欧，包括英国）。对夏令时启用地区才能使用。
time.asctime([tupletime])	接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"（2008年12月11日 周二18时07分14秒）的24个字符的字符串。
time.clock( )	用以浮点数计算的秒数返回当前的CPU时间。用来衡量不同程序的耗时，比time.time()更有用。
time.ctime([secs])	作用相当于asctime(localtime(secs))，未给参数相当于asctime()
time.gmtime([secs])	接收时间辍（1970纪元后经过的浮点秒数）并返回格林威治天文时间下的时间元组t。注：t.tm_isdst始终为0
time.localtime([secs])	接收时间辍（1970纪元后经过的浮点秒数）并返回当地时间下的时间元组t（t.tm_isdst可取0或1，取决于当地当时是不是夏令时）。
time.mktime(tupletime)	接受时间元组并返回时间辍（1970纪元后经过的浮点秒数）。
time.sleep(secs)	推迟调用线程的运行，secs指秒数。
time.strftime(fmt[,tupletime])	接收以时间元组，并返回以可读字符串表示的当地时间，格式由fmt决定。
time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')	根据fmt的格式把一个时间字符串解析为时间元组。
time.time( )	返回当前时间的时间戳（1970纪元后经过的浮点秒数）。
time.tzset()	根据环境变量TZ重新初始化时间相关设置。

Time模块包含了以下2个非常重要的属性：

属性	描述
time.timezone	属性time.timezone是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0大部分欧洲，亚洲，非洲）。
time.tzname	属性time.tzname包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。

日历（Calendar）模块

此模块的函数都是日历相关的，例如打印某月的字符月历。

星期一是默认的每周第一天，星期天是默认的最后一天。更改设置需调用calendar.setfirstweekday()函数。模块包含了以下内置函数：

序号	函数及描述
calendar.calendar(year,w=2,l=1,c=6)	返回一个多行字符串格式的year年年历，3个月一行，间隔距离为c。 每日宽度间隔为w字符。每行长度为21 *W+18+2* C。l是每星期行数。
calendar.firstweekday( )	返回当前每周起始日期的设置。默认情况下，首次载入caendar模块时返回0，即星期一。
calendar.isleap(year)	是闰年返回True，否则为false。
calendar.leapdays(y1,y2)	返回在Y1，Y2两年之间的闰年总数。
calendar.month(year,month,w=2,l=1)	返回一个多行字符串格式的year年month月日历，两行标题，一周一行。每日宽度间隔为w字符。每行的长度为7* w+6。l是每星期的行数。
calendar.monthcalendar(year,month)	返回一个整数的单层嵌套列表。每个子列表装载代表一个星期的整数。Year年month月外的日期都设为0;范围内的日子都由该月第几日表示，从1开始。
calendar.monthrange(year,month)	返回两个整数。第一个是该月的星期几的日期码，第二个是该月的日期码。日从0（星期一）到6（星期日）;月从1到12。
calendar.prcal(year,w=2,l=1,c=6)	相当于 print calendar.calendar(year,w,l,c).
calendar.prmonth(year,month,w=2,l=1)	相当于 print calendar.calendar（year，w，l，c）。
calendar.setfirstweekday(weekday)	设置每周的起始日期码。0（星期一）到6（星期日）。
calendar.timegm(tupletime)	和time.gmtime相反：接受一个时间元组形式，返回该时刻的时间辍（1970纪元后经过的浮点秒数）。
calendar.weekday(year,month,day)	返回给定日期的日期码。0（星期一）到6（星期日）。月份为 1（一月） 到 12（12月）。

其他相关模块和函数

在Python种，其他处理日期和时间的模块还有：

- datetime模块
- pytz模块

- ateutil模块

Python函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

- 函数代码块以def关键词开头，后接函数标识符名称和圆括号()。
- 任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- 函数内容以冒号起始，并且缩进。
- Return[expression]结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None。

语法

```
def functionname( parameters ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

实例

以下为一个简单的Python函数，它将一个字符串作为传入参数，再打印到标准显示设备上。

```
def printme( str ):
   "打印传入的字符串到标准显示设备上"
   print str
   return
```

函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从Python提示符执行。

如下实例调用了printme（）函数：

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

# Now you can call printme function
printme("我要调用用户自定义函数!");
printme("再次调用同一函数");
```

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

按值传递参数和按引用传递参数

所有参数（自变量）在Python里都是按引用传递。如果你在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。例如：

```
#!/usr/bin/python

# 可写函数说明
def changeme( mylist ):
   "修改传入的列表"
   mylist.append([1,2,3,4]);
   print "函数内取值: ", mylist
   return

# 调用changeme函数
mylist = [10,20,30];
changeme( mylist );
print "函数外取值: ", mylist
```

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

```
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]
```

参数

以下是调用函数时可使用的正式参数类型：

- 必备参数

- 命名参数
- 缺省参数
- 不定长参数

必备参数

必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

调用printme()函数，你必须传入一个参数，不然会出现语法错误：

```
#!/usr/bin/python

#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

#调用printme函数
printme();
```

以上实例输出结果：

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

命名参数

命名参数和函数调用关系紧密，调用方用参数的命名确定传入的参数值。你可以跳过不传的参数或者乱序传参，因为Python解释器能够用参数名匹配参数值。用命名参数调用printme()函数：

```
#!/usr/bin/python

#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

#调用printme函数
printme( str = "My string");
```

以上实例输出结果：

```
My string
```

下例能将命名参数顺序不重要展示得更清楚：

```
#!/usr/bin/python

#可写函数说明
def printinfo( name, age ):
   "打印任何传入的字符串"
   print "Name: ", name;
   print "Age ", age;
   return;

#调用printinfo函数
printinfo( age=50, name="miki" );
```

以上实例输出结果：

```
Name:  miki
Age  50
```

缺省参数

调用函数时，缺省参数的值如果没有传入，则被认为是默认值。下例会打印默认的age，如果age没有被传入：

```
#!/usr/bin/python

#可写函数说明
def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print "Name: ", name;
   print "Age ", age;
   return;

#调用printinfo函数
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

以上实例输出结果：

```
Name:  miki
Age  50
Name:  miki
Age  35
```

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述2种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可。如下实例：

```
#!/usr/bin/python

# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print "输出: "
   print arg1
   for var in vartuple:
      print var
   return;

# 调用printinfo 函数
printinfo( 10 );
printinfo( 70, 60, 50 );
```

以上实例输出结果：

```
输出:
10
输出:
70
60
50
```

匿名函数

用lambda关键词能创建小型匿名函数。这种函数得名于省略了用def声明函数的标准步骤。

- Lambda函数能接收任何数量的参数但只能返回一个表达式的值，同时只能不能包含命令或多个表达式。
- 匿名函数不能直接调用print，因为lambda需要一个表达式。
- lambda函数拥有自己的名字空间，且不能访问自有参数列表之外或全局名字空间里的参数。
- 虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

语法

lambda函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

如下实例：

```
#!/usr/bin/python

#可写函数说明
sum = lambda arg1, arg2: arg1 + arg2;

#调用sum函数
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

以上实例输出结果：

```
Value of total :  30
Value of total :  40
```

return语句

return语句[表达式]退出函数，选择性地向调用方返回一个表达式。不带参数值的return语句返回None。之前的例子都没有示范如何返回数值，下例便告诉你怎么做：

```
#!/usr/bin/python

# 可写函数说明
def sum( arg1, arg2 ):
   # 返回2个参数的和."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# 调用sum函数
total = sum( 10, 20 );
print "Outside the function : ", total
```

以上实例输出结果：

```
Inside the function :  30
Outside the function :  30
```

变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

- 全局变量
- 局部变量

变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

```
#!/usr/bin/python

total = 0; # This is global variable.
# 可写函数说明
def sum( arg1, arg2 ):
   #返回2个参数的和."
   total = arg1 + arg2; # total在这里是局部变量.
   print "Inside the function local total : ", total
   return total;

#调用sum函数
sum( 10, 20 );
print "Outside the function global total : ", total
```

以上实例输出结果：

```
Inside the function local total :  30
Outside the function global total :  0
```

Python 模块

模块让你能够有逻辑地组织你的Python代码段。

把相关的代码分配到一个 模块里能让你的代码更好用，更易懂。

模块也是Python对象，具有随机的名字属性用来绑定或引用。

简单地说，模块就是一个保存了Python代码的文件。模块能定义函数，类和变量。模块里也能包含可执行的代码。

例子

一个叫做aname的模块里的Python代码一般都能在一个叫aname.py的文件中找到。下例是个简单的模块support.py。

```
def print_func( par ):
   print "Hello : ", par
   return
```

import 语句

想使用Python源文件，只需在另一个源文件里执行import语句，语法如下：

```
import module1[, module2[,... moduleN]
```

当解释器遇到import语句，如果模块在当前的搜索路径就会被导入。

搜索路径是一个解释器会先进行搜索的所有目录的列表。如想要导入模块hello.py，需要把命令放在脚本的顶端：

```
#!/usr/bin/python

# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("Zara")
```

以上实例输出结果：

```
Hello : Zara
```

一个模块只会被导入一次，不管你执行了多少次import。这样可以防止导入模块被一遍又一遍地执行。

From…import 语句

Python的from语句让你从模块中导入一个指定的部分到当前命名空间中。语法如下：

```
from modname import name1[, name2[, ... nameN]]
```

例如，要导入模块fib的fibonacci函数，使用如下语句：

```
from fib import fibonacci
```

这个声明不会把整个fib模块导入到当前的命名空间中，它只会将fib里的fibonacci单个引入到执行这个声明的模块的全局符号表。

From…import* 语句

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

```
from modname import *
```

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。

定位模块

当你导入一个模块，Python解析器对模块位置的搜索顺序是：

- 当前目录
- 如果不在当前目录，Python则搜索在shell变量PYTHONPATH下的每个目录
- 如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/

模块搜索路径存存储在system模块的sys.path变量中。变量里包含当前目录，PYTHONPATH和由安装过程决定的默认目录。

PYTHONPATH变量

作为环境变量，PYTHONPATH由装在一个列表里的许多目录组成。PYTHONPATH的语法和shell变量PATH的一样。

在Windows系统，典型的PYTHONPATH如下：

```
set PYTHONPATH=c:\python20\lib;
```

在UNIX系统，典型的PYTHONPATH如下：

```
set PYTHONPATH=/usr/local/lib/python
```

命名空间和作用域

变量是拥有匹配对象的名字（标识符）。命名空间是一个包含了变量名称们（键）和它们各自相应的对象们（值）的字典。

一个Python表达式可以访问局部命名空间和全局命名空间里的变量。如果一个局部变量和一个全局变量重名，则局部变量会覆盖全局变量。

每个函数都有自己的命名空间。类的方法的作用域规则和通常函数的一样。

Python会智能地猜测一个变量是局部的还是全局的，它假设任何在函数内赋值的变量都是局部的。

因此，如果要给全局变量在一个函数里赋值，必须使用global语句。

global VarName的表达式会告诉Python， VarName是一个全局变量，这样Python就不会在局部命名空间里寻找这个变量了。

例如，我们在全局命名空间里定义一个变量money。我们再在函数内给变量money赋值，然后Python会假定money是一个局部变量。然而，我们并没有在访问前声明一个局部变量money，结果就是会出现一个UnboundLocalError的错误。取消global语句的注释就能解决这个问题。

```
#!/usr/bin/python

Money = 2000
def AddMoney():
   # 想改正代码就取消以下注释:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```

dir()函数

dir()函数一个排好序的字符串列表，内容是一个模块里定义过的名字。

返回的列表容纳了在一个模块里定义的所有模块，变量和函数。如下一个简单的实例：

```
#!/usr/bin/python

# 导入内置math模块
import math

content = dir(math)

print content;
```

以上实例输出结果：

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

在这里，特殊字符串变量**name**指向模块的名字，**file**指向该模块的导入文件名。

globals()和locals()函数

根据调用地方的不同，globals()和locals()函数可被用来返回全局和局部命名空间里的名字。

如果在函数内部调用locals()，返回的是所有能在该函数里访问的命名。

如果在函数内部调用globals()，返回的是所有在该函数里能访问的全局名字。

两个函数的返回类型都是字典。所以名字们能用keys()函数摘取。

reload()函数

当一个模块被导入到一个脚本，模块顶层部分的代码只会被执行一次。

因此，如果你想重新执行模块里顶层部分的代码，可以用reload()函数。该函数会重新导入之前导入过的模块。语法如下：

```
reload(module_name)
```

在这里，module_name要直接放模块的名字，而不是一个字符串形式。比如想重载hello模块，如下：

```
reload(hello)
```

Python中的包

包是一个分层次的文件目录结构，它定义了一个由模块及子包，和子包下的子包等组成的Python的应用环境。

考虑一个在Phone目录下的pots.py文件。这个文件有如下源代码：

```
#!/usr/bin/python

def Pots():
   print "I'm Pots Phone"
```

同样地，我们有另外两个保存了不同函数的文件：

- Phone/Isdn.py 含有函数Isdn()
- Phone/G3.py 含有函数G3()

现在，在Phone目录下创建file **init**.py：

- Phone/**init**.py

当你导入Phone时，为了能够使用所有函数，你需要在**init**.py里使用显式的导入语句，如下：

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

当你把这些代码添加到**init**.py之后，导入Phone包的时候这些类就全都是可用的了。

```
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

以上实例输出结果：

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

如上，为了举例，我们只在每个文件里放置了一个函数，但其实你可以放置许多函数。你也可以在这些文件里定义Python的类，然后为这些类建一个包。

Python 文件I/O

本章只讲述所有基本的的I/O函数，更多函数请参考Python标准文档。

打印到屏幕

最简单的输出方法是用print语句，你可以给它传递零个或多个用逗号隔开的表达式。此函数把你传递的表达式转换成一个字符串表达式，并将结果写到标准输出如下：

```
#!/usr/bin/python

print "Python is really a great language,", "isn't it?";
```

你的标准屏幕上会产生以下结果：

```
Python is really a great language, isn't it?
```

读取键盘输入

Python提供了两个内置函数从标准输入读入一行文本，默认的标准输入是键盘。如下：

- raw_input
- input

raw_input函数

raw_input([prompt]) 函数从标准输入读取一个行，并返回一个字符串（去掉结尾的换行符）：

```
#!/usr/bin/python

str = raw_input("Enter your input: ");
print "Received input is : ", str
```

这将提示你输入任意字符串，然后在屏幕上显示相同的字符串。当我输入"Hello Python！"，它的输出如下：

```
Enter your input: Hello Python
Received input is :  Hello Python
```

input函数

input([prompt]) 函数和raw_input([prompt]) 函数基本可以互换，但是input会假设你的输入是一个有效的Python表达式，并返回运算结果。

```
#!/usr/bin/python

str = input("Enter your input: ");
print "Received input is : ", str
```

这会产生如下的对应着输入的结果：

```
Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is :  [10, 20, 30, 40]
```

打开和关闭文件

到现在为止，您已经可以向标准输入和输进行读写。现在，来看看怎么读写实际的数据文件。

Python提供了必要的函数和方法进行默认情况下的文件基本操作。你可以用file对象做大部分的文件操作。

open函数

你必须先用Python内置的open()函数打开一个文件，创建一个file对象，相关的辅助方法才可以调用它进行读写。

语法：

```
file object = open(file_name [, access_mode][, buffering])
```

各个参数的细节如下：

- file_name：file_name变量是一个包含了你要访问的文件名称的字符串值。
- access_mode：access_mode决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读(r)。
- buffering:如果buffering的值被设为0，就不会有寄存。如果buffering的值取1，访问文件时会寄存行。如果将buffering的值设为大于1的整数，表明了这就是的寄存区的缓冲大小。如果取负值，寄存区的缓冲大小则为系统默认。

不同模式打开文件的完全列表：

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

File对象的属性

一个文件被打开后，你有一个file对象，你可以得到有关该文件的各种信息。

以下是和file对象相关的所有属性的列表：

属性	描述
file.closed	返回true如果文件已被关闭，否则返回false。
file.mode	返回被打开文件的访问模式。
file.name	返回文件的名称。
file.softspace	如果用print输出后，必须跟一个空格符，则返回false。否则返回true。

如下实例：

```
#!/usr/bin/python

# 打开一个文件
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

以上实例输出结果：

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

Close()方法

File对象的close（）方法刷新缓冲区里任何还没写入的信息，并关闭该文件，这之后便不能再进行写入。

当一个文件对象的引用被重新指定给另一个文件时，Python会关闭之前的文件。用close（）方法关闭文件是一个很好的习惯。

语法：

```
fileObject.close();
```

例子：

```
#!/usr/bin/python

# 打开一个文件
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
Name of the file:  foo.txt
```

读写文件：

file对象提供了一系列方法，能让我们的文件访问更轻松。来看看如何使用read()和write()方法来读取和写入文件。

Write()方法

Write()方法可将任何字符串写入一个打开的文件。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。

Write()方法不在字符串的结尾不添加换行符('\n')：

语法：

```
fileObject.write(string);
```

在这里，被传递的参数是要写入到已打开文件的内容。

例子：

```
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n");

# 关闭打开的文件
fo.close()
```

上述方法会创建foo.txt文件，并将收到的内容写入该文件，并最终关闭文件。如果你打开这个文件，将看到以下内容:

```
Python is a great language.
Yeah its great!!
```

read()方法

read（）方法从一个打开的文件中读取一个字符串。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。

语法：

```
fileObject.read([count]);
```

在这里，被传递的参数是要从已打开文件中读取的字节计数。该方法从文件的开头开始读入，如果没有传入count，它会尝试尽可能多地读取更多的内容，很可能是直到文件的末尾。

例子：

就用我们上面创建的文件foo.txt。

```
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
Read String is :  Python is
```

文件位置：

Tell()方法告诉你文件内的当前位置；换句话说，下一次的读写会发生在文件开头这么多字节之后：

seek（offset [,from]）方法改变当前文件的位置。Offset变量表示要移动的字节数。From变量指定开始移动字节的参考位置。

如果from被设为0，这意味着将文件的开头作为移动字节的参考位置。如果设为1，则使用当前的位置作为参考位置。如果它被设为2，那么该文件的末尾将作为参考位置。

例子：

就用我们上面创建的文件foo.txt。

```
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str

# 查找当前位置
position = fo.tell();
print "Current file position : ", position

# 把指针再次重新定位到文件开头
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

重命名和删除文件

Python的os模块提供了帮你执行文件处理操作的方法，比如重命名和删除文件。

要使用这个模块，你必须先导入它，然后可以调用相关的各种功能。

rename()方法：

rename()方法需要两个参数，当前的文件名和新文件名。

语法：

```
os.rename(current_file_name, new_file_name)
```

例子：

下例将重命名一个已经存在的文件test1.txt。

```
#!/usr/bin/python
import os

# 重命名文件test1.txt到test2.txt。
os.rename( "test1.txt", "test2.txt" )
```

remove()方法

你可以用remove()方法删除文件，需要提供要删除的文件名作为参数。

语法：

```
os.remove(file_name)
```

例子：

下例将删除一个已经存在的文件test2.txt。

```
#!/usr/bin/python
import os

# 删除一个已经存在的文件test2.txt
os.remove("text2.txt")
```

Python里的目录：

所有文件都包含在各个不同的目录下，不过Python也能轻松处理。os模块有许多方法能帮你创建，删除和更改目录。

mkdir()方法

可以使用os模块的mkdir()方法在当前目录下创建新的目录们。你需要提供一个包含了要创建的目录名称的参数。

语法：

```
os.mkdir("newdir")
```

例子：

下例将在当前目录下创建一个新目录test。

```
#!/usr/bin/python
import os

# 创建目录test
os.mkdir("test")
```

chdir()方法

可以用chdir()方法来改变当前的目录。chdir()方法需要的一个参数是你想设成当前目录的目录名称。

语法：

```
os.chdir("newdir")
```

例子：

下例将进入"/home/newdir"目录。

```
#!/usr/bin/python
import os

# 将当前目录改为"/home/newdir"
os.chdir("/home/newdir")
```

getcwd()方法：

getcwd()方法显示当前的工作目录。

语法：

```
os.getcwd()
```

例子：

下例给出当前目录：

```
#!/usr/bin/python
import os

# 给出当前的目录
os.getcwd()
```

rmdir()方法

rmdir()方法删除目录，目录名称以参数传递。

在删除这个目录之前，它的所有内容应该先被清除。

语法：

```
os.rmdir('dirname')
```

例子：

以下是删除" /tmp/test"目录的例子。目录的完全合规的名称必须被给出，否则会在当前目录下搜索该目录。

```
#!/usr/bin/python
import os

# 删除”/tmp/test”目录
os.rmdir( "/tmp/test"  )
```

文件、目录相关的方法

三个重要的方法来源能对Windows和Unix操作系统上的文件及目录进行一个广泛且实用的处理及操控，如下：

- File 对象方法: file对象提供了操作文件的一系列方法。
- OS 对象方法: 提供了处理文件及目录的一系列方法。

Python 异常处理

python提供了两个非常重要的功能来处理python程序在运行中出现的异常和错误。你可以使用该功能来调试python程序。

- 异常处理: 本站Python教程会具体介绍。
- 断言(Assertions):本站Python教程会具体介绍。

python标准异常

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
SystemExit	Python 解释器请求退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
KeyboardInterrupt	用户中断执行(通常是输入^C)

LookupError	无效数据查询的基类
IndexError	序列中没有没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

什么是异常？

异常即是一个事件，该事件会在程序执行过程中发生，影响了程序的正常执行。

一般情况下，在Python无法正常处理程序时就会发生一个异常。

异常是Python对象，表示一个错误。

当Python脚本发生异常时我们需要捕获处理它，否则程序会终止执行。

异常处理

捕捉异常可以使用try/except语句。

try/except语句用来检测try语句块中的错误，从而让except语句捕获异常信息并处理。

如果你不想在异常发生时结束你的程序，只需在try里捕获它。

语法：

以下为简单的*try....except...else*的语法：

```
try:
<语句>        #运行别的代码
except <名字>：
<语句>        #如果在try部份引发了'name'异常
except <名字>，<数据>:
<语句>        #如果引发了'name'异常，获得附加的数据
else:
<语句>        #如果没有异常发生
```

try的工作原理是，当开始一个try语句后，python就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try子句先执行，接下来会发生什么依赖于执行时是否出现异常。

- 如果当try后的语句执行时发生异常，python就跳回到try并执行第一个匹配该异常的except子句，异常处理完毕，控制流就通过整个try语句（除非在处理异常时又引发新的异常）。
- 如果在try后的语句里发生了异常，却没有匹配的except子句，异常将被递交到上层的try，或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。
- 如果在try子句执行时没有发生异常，python将执行else语句后的语句（如果有else的话），然后控制流通过整个try语句。

实例

下面是简单的例子，它打开一个文件，在该文件中的内容写入内容，且并未发生异常：

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

以上程序输出结果：

```
Written content in the file successfully
```

实例

下面是简单的例子，它打开一个文件，在该文件中的内容写入内容，但文件没有写入权限，发生了异常：

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

以上程序输出结果：

```
Error: can't find file or read data
```

使用except而不带任何异常类型

你可以不带任何异常类型使用except，如下实例：

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

以上方式try-except语句捕获所有发生的异常。但这不是一个很好的方式，我们不能通过该程序识别出具体的异常信息。因为它捕获所有的异常。

使用except而带多种异常类型

你也可以使用相同的except语句来处理多个异常信息，如下所示：

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

try-finally 语句

try-finally 语句无论是否发生异常都将执行最后的代码。

```
try:
<语句>
finally:
<语句>    #退出try时总会执行
raise
```

注意：你可以使用except语句或者finally语句，但是两者不能同时使用。else语句也不能与finally语句同时使用

实例

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

如果打开的文件没有可写权限，输出如下所示：

```
Error: can't find file or read data
```

同样的例子也可以写成如下方式：

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

当在try块中抛出一个异常，立即执行finally块代码。

finally块中的所有语句执行后，异常被再次提出，并执行except块代码。

参数的内容不同于异常。

异常的参数

一个异常可以带上参数，可作为输出的异常信息参数。

你可以通过except语句来捕获异常的参数，如下所示：

```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

变量接收的异常值通常包含在异常的语句中。在元组的表单中变量可以接收一个或者多个值。

元组通常包含错误字符串，错误数字，错误位置。

实例

以下为单个异常的实例：

```
#!/usr/bin/python

# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError, Argument:
      print "The argument does not contain numbers\n", Argument

# Call above function here.
temp_convert("xyz");
```

以上程序执行结果如下：

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

触发异常

我们可以使用raise语句自己触发异常

raise语法格式如下：

```
raise [Exception [, args [, traceback]]]
```

语句中Exception是异常的类型（例如，NameError）参数是一个异常参数值。该参数是可选的，如果不提供，异常的参数是"None"。

最后一个参数是可选的（在实践中很少使用），如果存在，是跟踪异常对象。

实例

一个异常可以是一个字符串，类或对象。 Python的内核提供的异常，大多数都是实例化的类，这是一个类的实例的参数。

定义一个异常非常简单，如下所示：

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

注意：为了能够捕获异常，"except"语句必须有用相同的异常来抛出类对象或者字符串。

例如我们捕获以上异常，"except"语句如下所示：

```
try:
   Business Logic here...
except "Invalid level!":
   Exception handling here...
else:
   Rest of the code here...
```

用户自定义异常

通过创建一个新的异常类，程序可以命名它们自己的异常。异常应该是典型的继承自Exception类，通过直接或间接的方式。

以下为与RuntimeError相关的实例,实例中创建了一个类，基类为RuntimeError，用于在异常触发时输出更多的信息。

在try语句块中，用户自定义的异常后执行except块语句，变量 e 是用于创建Networkerror类的实例。

```
class Networkerror(RuntimeError):
   def __init__(self, arg):
      self.args = arg
```

在你定义以上类后，你可以触发该异常，如下所示：

```
try:
   raise Networkerror("Bad hostname")
except Networkerror,e:
   print e.args
```

Python 高级教程

Python面向对象

Python从设计之初就已经是一门面向对象的语言，正因为如此，在Python中创建一个类和对象是很容易的。本章节我们将详细介绍Python的面向对象编程。

如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习Python的面向对象编程。

接下来我们先来简单的了解下面向对象的一些基本特征。

面向对象技术简介

- 类**(Class):** 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- 类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量用于处理类及其实例对象的相关的数据。
- 方法重载：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重载。
- 实例变量：定义在方法中的变量，只作用于当前实例的类。
- 继承：即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个Dog类型的对象派生自Animal类，这是模拟"是一个（is-a）"关系（例图，Dog是一个Animal）。
- 实例化：创建一个类的实例，类的具体对象。
- 方法：类中定义的函数。
- 对象：通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

创建类

使用class语句来创建一个新类，class之后为类的名称并以冒号结尾，如下实例:

```
class ClassName:
   'Optional class documentation string'#类文档字符串
   class_suite  #类体
```

类的帮助信息可以通过ClassName.**doc**查看。

class_suite 由类成员，方法，数据属性组成。

实例

以下是一个简单的Python类实例:

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- empCount变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用Employee.empCount访问。
- 第一种方法**init**()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法

创建实例对象

要创建一个类的实例，你可以使用类的名称，并通过**init**方法接受参数。

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

访问属性

您可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量:

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

完整实例：

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

执行以上代码输出结果如下：

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

你可以添加，删除，修改类的属性，如下所示：

```
emp1.age = 7  # 添加一个 'age' 属性
emp1.age = 8  # 修改 'age' 属性
del emp1.age  # 删除 'age' 属性
```

你也可以使用以下函数的方式来访问属性：

- getattr(obj, name[, default]) : 访问对象的属性。
- hasattr(obj,name) : 检查是否存在一个属性。
- setattr(obj,name,value) : 设置一个属性。如果属性不存在，会创建一个新属性。
- delattr(obj, name) : 删除属性。

```
hasattr(emp1, 'age')    # 如果存在 'age' 属性返回 True。
getattr(emp1, 'age')    # 返回 'age' 属性的值
setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8
delattr(empl, 'age')    # 删除属性 'age'
```

Python内置类属性

- **dict** : 类的属性（包含一个字典，由类的数据属性组成）
- **doc** :类的文档字符串

- **name**: 类名
- **module**: 类定义所在的模块（类的全名是'**main**.className'，如果类位于一个导入模块mymod中，那么className.**module** 等于 mymod）
- **bases** : 类的所有父类构成元素（包含了以个由所有父类组成的元组）

Python内置类属性调用实例如下：

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

执行以上代码输出结果如下：

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

python对象销毁(垃圾回收)

同Java语言一样，Python使用了引用计数这一简单技术来追踪内存中的对象。

在Python内部记录着所有使用中的对象各有多少引用。

一个内部跟踪变量，称为一个引用计数器。

当对象被创建时， 就创建了一个引用计数， 当这个对象不再需要时， 也就是说， 这个对象的引用计数变为0 时， 它被垃圾回收。但是回收不是"立即"的， 由解释器在适当的时机，将垃圾对象占用的内存空间回收。

```
a = 40      # 创建对象  <40>
b = a       # 增加引用， <40> 的计数
c = [b]     # 增加引用.  <40> 的计数

del a       # 减少引用 <40> 的计数
b = 100     # 减少引用 <40> 的计数
c[0] = -1   # 减少引用 <40> 的计数
```

垃圾回收机制不仅针对引用计数为0的对象，同样也可以处理循环引用的情况。循环引用指的是，两个对象相互引用，但是没有其他变量引用他们。这种情况下，仅使用引用计数是不够的。Python 的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。作为引用计数的补充，垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象。在这种情况下，解释器会暂停下来，试图清理所有未引用的循环。

实例

析构函数 **del**，**del**在对象消逝的时候被调用，当对象不再被使用时，**del**方法运行：

```
#!/usr/bin/python

class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # 打印对象的id
del pt1
del pt2
del pt3
<pre>
<p>以上实例运行结果如下：</p>
<pre>
3083401324 3083401324 3083401324
Point destroyed
```

注意：通常你需要在单独的文件中定义一个类，

类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。

需要注意的地方：继承语法 class 派生类名（基类名）：//... 基类名写作括号里，基本类是在类定义的时候，在元组之中指明的。

在python中继承中的一些特点：

- 1：在继承中基类的构造（**init**()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
- 2：在调用基类的方法时，需要加上基类的类名前缀，且需要带上self参数变量。区别于在类中调用普通函数时并不需要带上self参数
- 3：Python总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。（先在本类中查找调用的方法，找不到才去基类中找）。

如果在继承元组中列了一个以上的类，那么它就被称作"多重继承" 。

语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

实例：

```
#!/usr/bin/python

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # 实例化子类
c.childMethod()      # 调用子类的方法
c.parentMethod()     # 调用父类方法
c.setAttr(200)       # 再次调用父类的方法
c.getAttr()          # 再次调用父类的方法
```

以上代码执行结果如下：

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

你可以继承多个类

```
class A:        # define your class A
.....

class B:         # define your calss B
.....

class C(A, B):   # subclass of A and B
.....
```

你可以使用issubclass()或者isinstance()方法来检测。

- issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)
- isinstance(obj, Class) 布尔函数如果obj是Class类的实例对象或者是一个Class子类的实例对象则返回true。

重载方法

如果你的父类方法的功能不能满足你的需求，你可以在子类重载你父类的方法：

实例：

```
#!/usr/bin/python

class Parent:        # 定义父类
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # 定义子类
   def myMethod(self):
      print 'Calling child method'

c = Child()          # 子类实例
c.myMethod()         # 子类调用重载方法
```

执行以上代码输出结果如下：

```
Calling child method
```

基础重载方法

下表列出了一些通用的功能，你可以在自己的类重写：

方法	描述 & 简单的调用
__init__ (self [,args...])	构造函数 简单的调用方法: *obj = className(args)*
__del__(self)	析构方法, 删除一个对象 简单的调用方法 : *del obj*
__repr__(self)	转化为供解释器读取的形式 简单的调用方法 : *repr(obj)*
__str__(self)	用于将值转化为适于人阅读的形式 简单的调用方法 : *str(obj)*
__cmp__ (self, x)	对象比较 简单的调用方法 : *cmp(obj, x)*

运算符重载

Python同样支持运算符重载，实例如下：

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

以上代码执行结果如下所示:

```
Vector(7,8)
```

隐藏数据

在python中实现数据隐藏很简单，不需要在前面加什么关键字，只要把类变量名或成员函数前面加两个下划线即可实现数据隐藏的功能，这样，对于类的实例来说，其变量名和成员函数是不能使用的，对于其类的继承类来说，也是隐藏的，这样，其继承类可以定义其一模一样的变量名或成员函数名，而不会引起命名冲突。 实例：

```
#!/usr/bin/python

class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

Python 通过改变名称来包含类名:

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python不允许实例化的类访问隐藏数据，但你可以使用object._className__attrName访问属性，将如下代码替换以上代码的最后一行代码：

```
.........................
print counter._JustCounter__secretCount
```

执行以上代码，执行结果如下：

```
1
2
2
```

Python正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。Python 自1.5版本起增加了re 模块，它提供 Perl 风格的正则表达式模式。

re 模块使 Python 语言拥有全部的正则表达式功能。

compile 函数根据一个模式字符串和可选的标志参数生成一个正则表达式对象。该对象拥有一系列方法用于正则表达式匹配和替换。

re 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串做为它们的第一个参数。

本章节主要介绍Python中常用的正则表达式处理函数。

re.match函数

re.match 尝试从字符串的开始匹配一个模式。

函数语法：

```
re.match(pattern, string, flags=0)
```

函数参数说明：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

匹配成功re.match方法返回一个匹配的对象，否则返回None。

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

匹配对象方法	描述
group(num=0)	匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。
groups()	返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

以上实例执行结果如下：

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

re.search方法

re.match 尝试从字符串的开始匹配一个模式。

函数语法：

```
re.search(pattern, string, flags=0)
```

函数参数说明：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

匹配成功re.search方法方法返回一个匹配的对象，否则返回None。

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

匹配对象方法	描述
group(num=0)	匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。
groups()	返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

以上实例执行结果如下：

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

re.match与re.search的区别

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"

matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
   print "search --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"
```

以上实例运行结果如下：

```
No match!!
search --> matchObj.group() :  dogs
```

检索和替换

Python 的re模块提供了re.sub用于替换字符串中的匹配项。

语法：

```
re.sub(pattern, repl, string, max=0)
```

返回的字符串是在字符串中用 RE 最左边不重复的匹配来替换。如果模式没有发现，字符将被没有改变地返回。

可选参数 count 是模式匹配后替换的最大次数；count 必须是非负整数。缺省值是 0 表示替换所有的匹配。

实例：

```
#!/usr/bin/python
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*/o>, "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

以上实例执行结果如下：

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

正则表达式修饰符 - 可选标志

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR(|) 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志：

修饰符	描述
re.I	使匹配对大小写不敏感
re.L	做本地化识别（locale-aware）匹配
re.M	多行匹配，影响 ^ 和 $
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B.
re.X	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。

正则表达式模式

模式字符串使用特殊的语法来表示一个正则表达式：

字母和数字表示他们自身。一个正则表达式模式中的字母和数字匹配同样的字符串。

多数字母和数字前加一个反斜杠时会拥有不同的含义。

标点符号只有被转义时才匹配自身，否则它们表示特殊的含义。

反斜杠本身需要使用反斜杠转义。

由于正则表达式通常都包含反斜杠，所以你最好使用原始字符串来表示它们。模式元素(如 r'/t'，等价于'//t')匹配相应的特殊字符。

下表列出了正则表达式模式语法中的特殊元素。如果你使用模式的同时提供了可选的标志参数，某些模式元素的含义会改变。

模式	描述
^	匹配字符串的开头
$	匹配字符串的末尾。
.	匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。
[...]	用来表示一组字符,单独列出：[amk] 匹配 'a'，'m'或'k'
...	不在[]中的字符：abc 匹配除了a,b,c之外的字符。
re*	匹配0个或多个的表达式。
re+	匹配1个或多个的表达式。
re?	匹配0个或1个由前面的正则表达式定义的片段，贪婪方式
re{ n}	
re{ n,}	精确匹配n个前面表达式。
re{ n, m}	匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式
a | b	匹配a或b
(re)	G匹配括号内的表达式，也表示一个组
(?imx)	正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。
(?-imx)	正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。
(?: re)	类似 (...), 但是不表示一个组
(?	

imx: re)	在括号中使用i, m, 或 x 可选标志
(?-imx: re)	在括号中不使用i, m, 或 x 可选标志
(?#...)	注释.
(?= re)	前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。
(?! re)	前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功
(?> re)	匹配的独立模式，省去回溯。
\w	匹配字母数字
\W	匹配非字母数字
\s	匹配任意空白字符，等价于 [\t\n\r\f].
\S	匹配任意非空字符
\d	匹配任意数字，等价于 [0-9].
\D	匹配任意非数字
\A	匹配字符串开始
\Z	匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。c
\z	匹配字符串结束
\G	匹配最后匹配完成的位置。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'， 但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\n, \t, 等.	匹配一个换行符。匹配一个制表符。等
\1...\9	比赛第n个分组的子表达式。
\10	匹配第n个分组的子表达式，如果它经匹配。否则指的是八进制字符码的表达式。

正则表达式实例

字符匹配

实例	描述
python	匹配 "python".

字符类

实例	描述
[Pp]ython	匹配 "Python" 或 "python"
rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配中括号内的任意一个字母
[0-9]	匹配任何数字。类似于 [0123456789]
[a-z]	匹配任何小写字母
[A-Z]	匹配任何大写字母
[a-zA-Z0-9]	匹配任何字母及数字
aeiou	除了aeiou字母以外的所有字符
0-9	匹配除了数字外的字符

特殊字符类

实例	描述
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 0-9。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 \f\n\r\t\v。
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'。
\W	匹配任何非单词字符。等价于 'A-Za-z0-9_'。

Python CGI编程

什么是CGI

CGI 目前由NCSA维护，NCSA定义CGI如下：

CGI(Common Gateway Interface),通用网关接口,它是一段程序,运行在服务器上如：HTTP服务器，提供同客户端HTML页面的接口。

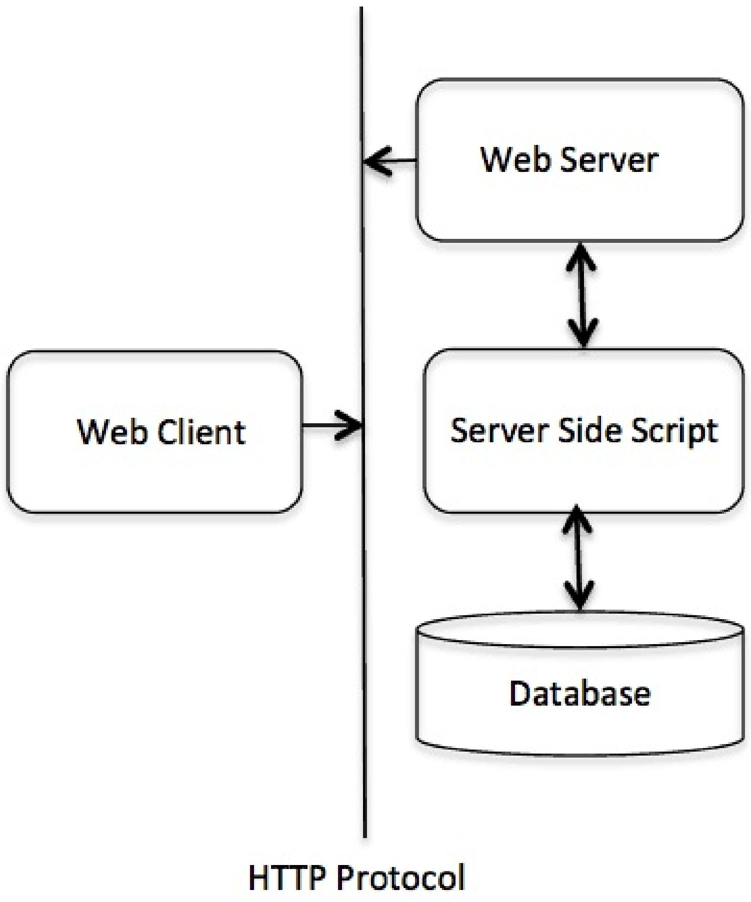
网页浏览

为了更好的了解CGI是如何工作的，我们可以从在网页上点击一个链接或URL的流程：

- 1、使用你的浏览器访问URL并连接到HTTP web 服务器。
- 2、Web服务器接收到请求信息后会解析URL，并查找访问的文件在服务器上是否存在，如果存在返回文件的内容，否则返回错误信息。
- 3、浏览器从服务器上接收信息，并显示接收的文件或者错误信息。

CGI程序可以是Python脚本，PERL脚本，SHELL脚本，C或者C++程序等。

CGI架构图



Web服务器支持及配置

在你进行CGI编程前，确保您的Web服务器支持CGI及已经配置了CGI的处理程序。

所有的HTTP服务器执行CGI程序都保存在一个预先配置的目录。这个目录被称为CGI目录，并按照惯例，它被命名为/var/www/cgi-bin目录。

CGI文件的扩展名为.cgi，python也可以使用.py扩展名。

默认情况下，Linux服务器配置运行的cgi-bin目录中为/var/www。

如果你想指定其他运行CGI脚本的目录，可以修改httpd.conf配置文件，如下所示：

```
<Directory "/var/www/cgi-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>

<Directory "/var/www/cgi-bin">
Options All
</Directory>
```

第一个CGI程序

我们使用Python创建第一个CGI程序，文件名为hellp.py，文件位于/var/www/cgi-bin目录中，内容如下，修改文件的权限为755：

```
#!/usr/bin/python

print "Content-type:text/html\r\n\r\n"
print '<html>'
print '<head>'
print '<title>Hello Word - First CGI Program</title>'
print '</head>'
print '<body>'
print '<h2>Hello Word! This is my first CGI program</h2>'
print '</body>'
print '</html>'
```

以上程序在浏览器访问显示结果如下：

```
Hello Word! This is my first CGI program
```

这个的hello.py脚本是一个简单的Python脚本，脚本第一的输出内容"Content-type:text/html\r\n\r\n"发送到浏览器并告知浏览器显示的内容类型为"text/html"。

HTTP头部

hello.py文件内容中的" Content-type:text/html\r\n\r\n"即为HTTP头部的一部分，它会发送给浏览器告诉浏览器文件的内容类型。

HTTP头部的格式如下：

```
HTTP 字段名: 字段内容

例如
Content-type: text/html\r\n\r\n
```

以下表格介绍了CGI程序中HTTP头部经常使用的信息：

头	描述
Content-type:	请求的与实体对应的MIME信息。例如: Content-type:text/html
Expires: Date	响应过期的日期和时间
Location: URL	用来重定向接收方到非请求URL的位置来完成请求或标识新的资源
Last-modified: Date	请求资源的最后修改时间
Content-length: N	请求的内容长度
Set-Cookie: String	设置Http Cookie

CGI环境变量

所有的CGI程序都接收以下的环境变量，这些变量在CGI程序中发挥了重要的作用：

变量名	描述
CONTENT_TYPE	这个环境变量的值指示所传递来的信息的MIME类型。目前，环境变量CONTENT_TYPE一般都是：application/x-www-form-urlencoded,他表示数据来自于HTML表单。
CONTENT_LENGTH	如果服务器与CGI程序信息的传递方式是POST，这个环境变量即使从标准输入STDIN中可以读到的有效数据的字节数。这个环境变量在读取所输入的数据时必须使用。
HTTP_COOKIE	客户机内的 COOKIE 内容。
HTTP_USER_AGENT	提供包含了版本数或其他专有数据的客户浏览器信息。
PATH_INFO	这个环境变量的值表示紧接在CGI程序名之后的其他路径信息。它常常作为CGI程序的参数出现。
QUERY_STRING	如果服务器与CGI程序信息的传递方式是GET，这个环境变量的值即使所传递的信息。这个信息经跟在CGI程序名的后面，两者中间用一个问号'?'分隔。
REMOTE_ADDR	这个环境变量的值是发送请求的客户机的IP地址，例如上面的192.168.1.67。这个值总是存在的。而且它是Web客户机需要提供给Web服务器的唯一标识，可以在CGI程序中用它来区分不同的Web客户机。
REMOTE_HOST	这个环境变量的值包含发送CGI请求的客户机的主机名。如果不支持你想查询，则无需定义此环境变量。
REQUEST_METHOD	提供脚本被调用的方法。对于使用 HTTP/1.0 协议的脚本，仅 GET 和 POST 有意义。
SCRIPT_FILENAME	CGI脚本的完整路径
SCRIPT_NAME	CGI脚本的的名称
SERVER_NAME	这是你的 WEB 服务器的主机名、别名或IP地址。
SERVER_SOFTWARE	这个环境变量的值包含了调用CGI程序的HTTP服务器的名称和版本号。例如，上面的值为Apache/2.2.14(Unix)

以下是一个简单的CGI脚本输出CGI的环境变量：

```
#!/usr/bin/python

import os

print "Content-type: text/html\r\n\r\n";
print "Environment<\br>";
for param in os.environ.keys():
  print "<b>%20s</b>: %s<\br>" % (param, os.environ[param])
```

GET和POST方法

浏览器客户端通过两种方法向服务器传递信息，这两种方法就是 GET 方法和 POST 方法。

使用GET方法传输数据

GET方法发送编码后的用户信息到服务端，数据信息包含在请求页面的URL上，以"?"号分割, 如下所示：

```
http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2
```

有关 GET 请求的其他一些注释：

- GET 请求可被缓存
- GET 请求保留在浏览器历史记录中
- GET 请求可被收藏为书签
- GET 请求不应在处理敏感数据时使用
- GET 请求有长度限制
- GET 请求只应当用于取回数据

简单的url实例：GET方法

以下是一个简单的URL，使用GET方法向hello_get.py程序发送两个参数：

```
/cgi-bin/hello_get.py?first_name=ZARA&last_name=ALI
```

以下为hello_get.py文件的代码：

```
#!/usr/bin/python

# CGI处理模块
import cgi, cgitb

# 创建 FieldStorage 的实例化
form = cgi.FieldStorage()

# 获取数据
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Hello - Second CGI Program</title>"
print "</head>"
print "<body>"
print "<h2>Hello %s %s</h2>" % (first_name, last_name)
print "</body>"
print "</html>"
```

浏览器请求输出结果：

```
Hello ZARA ALI
```

简单的表单实例：GET方法

以下是一个通过HTML的表单使用GET方法向服务器发送两个数据，提交的服务器脚本同样是hello_get.py文件，代码如下：

```
<form action="/cgi-bin/hello_get.py" method="get">
First Name: <input type="text" name="first_name">  <br />

Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
```

使用POST方法传递数据

使用POST方法向服务器传递数据是更安全可靠的，像一些敏感信息如用户密码等需要使用POST传输数据。

以下同样是hello_get.py ，它也可以处理浏览器提交的POST表单数据:

```
#!/usr/bin/python

# 引入 CGI 模块
import cgi, cgitb

# 创建 FieldStorage 实例
form = cgi.FieldStorage()

# 获取表单数据
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Hello - Second CGI Program</title>"
print "</head>"
print "<body>"
print "<h2>Hello %s %s</h2>" % (first_name, last_name)
print "</body>"
print "</html>"
```

以下为表单通过POST方法向服务器脚本hello_get.py提交数据:

```
<form action="/cgi-bin/hello_get.py" method="post">
First Name: <input type="text" name="first_name"><br />
Last Name: <input type="text" name="last_name" />

<input type="submit" value="Submit" />
</form>
```

通过CGI程序传递checkbox数据

checkbox用于提交一个或者多个选项数据，HTML代码如下：

```
<form action="/cgi-bin/checkbox.cgi" method="POST" target="_blank">
<input type="checkbox" name="maths" value="on" /> Maths
<input type="checkbox" name="physics" value="on" /> Physics
<input type="submit" value="Select Subject" />
</form>
```

以下为 checkbox.cgi 文件的代码：

```
#!/usr/bin/python

# 引入 CGI 处理模块
import cgi, cgitb

# 创建 FieldStorage的实例
form = cgi.FieldStorage()

# 接收字段数据
if form.getvalue('maths'):
   math_flag = "ON"
else:
   math_flag = "OFF"

if form.getvalue('physics'):
   physics_flag = "ON"
else:
   physics_flag = "OFF"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Checkbox - Third CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> CheckBox Maths is : %s</h2>" % math_flag
print "<h2> CheckBox Physics is : %s</h2>" % physics_flag
print "</body>"
print "</html>"
```

通过CGI程序传递Radio数据

Radio只向服务器传递一个数据，HTML代码如下：

```
<form action="/cgi-bin/radiobutton.py" method="post" target="_blank">
<input type="radio" name="subject" value="maths" /> Maths
<input type="radio" name="subject" value="physics" /> Physics
<input type="submit" value="Select Subject" />
</form>
```

radiobutton.py 脚本代码如下：

```
#!/usr/bin/python

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('subject'):
   subject = form.getvalue('subject')
else:
   subject = "Not set"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Radio - Fourth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"
```

通过CGI程序传递 Textarea 数据

Textarea向服务器传递多行数据，HTML代码如下：

```
<form action="/cgi-bin/textarea.py" method="post" target="_blank">
<textarea name="textcontent" cols="40" rows="4">
Type your text here...
</textarea>
<input type="submit" value="Submit" />
</form>
```

textarea.cgi脚本代码如下：

```
#!/usr/bin/python

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('textcontent'):
   text_content = form.getvalue('textcontent')
else:
   text_content = "Not entered"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>";
print "<title>Text Area - Fifth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Entered Text Content is %s</h2>" % text_content
print "</body>"
```

通过CGI程序传递下拉数据

HTML下拉框代码如下：

```
<form action="/cgi-bin/dropdown.py" method="post" target="_blank">
<select name="dropdown">
<option value="Maths" selected>Maths</option>
<option value="Physics">Physics</option>
</select>
<input type="submit" value="Submit"/>
</form>
```

dropdown.py 脚本代码如下所示：

```
#!/usr/bin/python

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('dropdown'):
   subject = form.getvalue('dropdown')
else:
   subject = "Not entered"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Dropdown Box - Sixth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"
```

CGI中使用Cookie

在http协议一个很大的缺点就是不作用户身份的判断，这样给编程人员带来很大的不便，

而cookie功能的出现弥补了这个缺憾。

所有cookie就是在客户访问脚本的同时，通过客户的浏览器，在客户硬盘上写入纪录数据，当下次客户访问脚本时取回数据信息，从而达到身份判别的功能，cookie常用在密码判断中。

cookie的语法

http cookie的发送是通过http头部来实现的，他早于文件的传递，头部set-cookie的语法如下：

```
Set-cookie:name=name;expires=date;path=path;domain=domain;secure
```

- name=name: 需要设置cookie的值(name不能使用"；"和"，"号),有多个name值时用"；"分隔例如：name1=name1;name2=name2;name3=name3。
- expires=date: cookie的有效期限,格式： expires="Wdy,DD-Mon-YYYY HH:MM:SS"
- path=path: 设置cookie支持的路径,如果path是一个路径，则cookie对这个目录下的所有文件及子目录生效，例如： path="/cgi-bin/"，如果path是一个文件，则cookie指对这个文件生效，例如：path="/cgi-bin/cookie.cgi"。
- domain=domain: 对cookie生效的域名，例如：domain="www.chinalb.com"
- secure: 如果给出此标志，表示cookie只能通过SSL协议的https服务器来传递。
- cookie的接收是通过设置环境变量HTTP_COOKIE来实现的，CGI程序可以通过检索该变量获取cookie信息。

Cookie设置

Cookie的设置非常简单，cookie会在http头部单独发送。以下实例在cookie中设置了UserID和 Password：

```
<pre>
#!/usr/bin/python

print "Set-Cookie:UserID=XYZ;\r\n"
print "Set-Cookie:Password=XYZ123;\r\n"
print "Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";\r\n"
print "Set-Cookie:Domain=www.w3cschool.cc;\r\n"
print "Set-Cookie:Path=/perl;\n"
print "Content-type:text/html\r\n\r\n"
...........Rest of the HTML Content....
```

以上实例使用了 Set-Cookie 头信息来设置Cookie信息，可选项中设置了Cookie的其他属性，如过期时间Expires，域名Domain，路径Path。这些信息设置在 "Content-type:text/html\r\n\r\n"之前。

检索Cookie信息

Cookie信息检索页非常简单，Cookie信息存储在CGI的环境变量HTTP_COOKIE中，存储格式如下：

```
key1=value1;key2=value2;key3=value3....
```

以下是一个简单的CGI检索cookie信息的程序：

```
#!/usr/bin/python

# Import modules for CGI handling 
from os import environ
import cgi, cgitb

if environ.has_key('HTTP_COOKIE'):
   for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
      (key, value ) = split(cookie, '=');
      if key == "UserID":
         user_id = value

      if key == "Password":
         password = value

print "User ID  = %s" % user_id
print "Password = %s" % password
```

以上脚本输出结果如下：

```
User ID = XYZ
Password = XYZ123
```

文件上传实例：

HTML设置上传文件的表单需要设置enctype 属性为multipart/form-data，代码如下所示：

```
<html>
<body>
   <form enctype="multipart/form-data" 
                     action="save_file.py" method="post">
   <p>File: <input type="file" name="filename" /></p>
   <p><input type="submit" value="Upload" /></p>
   </form>
</body>
</html>
```

save_file.py脚本文件代码如下：

```
#!/usr/bin/python

import cgi, os
import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# 获取文件名
fileitem = form['filename']

# 检测文件是否上传
if fileitem.filename:
   # 设置文件路径
   fn = os.path.basename(fileitem.filename)
   open('/tmp/' + fn, 'wb').write(fileitem.file.read())

   message = 'The file "' + fn + '" was uploaded successfully'

else:
   message = 'No file was uploaded'

print """\
Content-Type: text/html\n
<html>
<body>
   <p>%s</p>
</body>
</html>
""" % (message,)
```

如果你使用的系统是Unix/Linux，你必须替换文件分隔符，在window下只需要使用open()语句即可：

```
fn = os.path.basename(fileitem.filename.replace("\\", "/" ))
```

文件下载对话框

如果我们需要为用户提供文件下载链接，并在用户点击链接后弹出文件下载对话框，我们通过设置HTTP头信息来实现这些功能，功能代码如下：

```
#!/usr/bin/python

# HTTP Header
print "Content-Type:application/octet-stream; name=\"FileName\"\r\n";
print "Content-Disposition: attachment; filename=\"FileName\"\r\n\n";

# Actual File Content will go hear.
fo = open("foo.txt", "rb")

str = fo.read();
print str

# Close opend file
fo.close()
```

python操作mysql数据库

Python 标准数据库接口为 Python DB-API，Python DB-API为开发人员提供了数据库应用编程接口。

Python 数据库接口支持非常多的数据库，你可以选择适合你项目的数据库：

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

你可以访问Python数据库接口及API查看详细的支持数据库列表。

不同的数据库你需要下载不同的DB API模块，例如你需要访问Oracle数据库和Mysql数据，你需要下载Oracle和MySQL数据库模块。

DB-API 是一个规范. 它定义了一系列必须的对象和数据库存取方式, 以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口 。

Python的DB-API，为大多数的数据库实现了接口，使用它连接各数据库后，就可以用相同的方式操作各数据库。

Python DB-API使用流程：

- 引入 API 模块。
- 获取与数据库的连接。
- 执行SQL语句和存储过程。
- 关闭数据库连接。

什么是MySQLdb?

MySQLdb 是用于Python链接Mysql数据库的接口，它实现了 Python 数据库 API 规范 V2.0，基于 MySQL C API 上建立的。

如何安装MySQLdb?

为了用DB-API编写MySQL脚本，必须确保已经安装了MySQL。复制以下代码，并执行：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb
```

如果执行后的输出结果如下所示，意味着你没有安装 MySQLdb 模块：

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
```

安装MySQLdb，请访问 http://sourceforge.net/projects/mysql-python ，(Linux平台可以访问：https://pypi.python.org/pypi/MySQL-python)从这里可选择适合您的平台的安装包，分为预编译的二进制文件和源代码安装包。

如果您选择二进制文件发行版本的话，安装过程基本安装提示即可完成。如果从源代码进行安装的话，则需要切换到MySQLdb发行版本的顶级目录，并键入下列命令:

```
$ gunzip MySQL-python-1.2.2.tar.gz
$ tar -xvf MySQL-python-1.2.2.tar
$ cd MySQL-python-1.2.2
$ python setup.py build
$ python setup.py install
```

注意：请确保您有root权限来安装上述模块。

数据库连接

连接数据库前，请先确认以下事项：

- 您已经创建了数据库 TESTDB.
- 在TESTDB数据库中您已经创建了表 EMPLOYEE
- EMPLOYEE表字段为 FIRST_NAME, LAST_NAME, AGE, SEX 和 INCOME。
- 连接数据库TESTDB使用的用户名为 "testuser" ，密码为 "test123",你可以可以自己设定或者直接使用root用户名及其密码，Mysql数据库用户授权请使用Grant命令。
- 在你的机子上已经安装了 Python MySQLdb 模块。
- 如果您对sql语句不熟悉，可以访问我们的 SQL基础教程

实例：

以下实例链接Mysql的TESTDB数据库：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 使用execute方法执行SQL语句
cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取一条数据库。
data = cursor.fetchone()

print "Database version : %s " % data

# 关闭数据库连接
db.close()
```

执行以上脚本输出结果如下：

```
Database version : 5.0.45
```

创建数据库表

如果数据库连接存在我们可以使用execute()方法来为数据库创建表，如下所示创建表EMPLOYEE：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 如果数据表已经存在使用 execute() 方法删除表。
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 创建数据表SQL语句
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""

cursor.execute(sql)

# 关闭数据库连接
db.close()
```

数据库插入操作

以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# 关闭数据库连接
db.close()
```

以上例子也可以写成如下形式：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

实例：

以下代码使用变量向SQL语句中传递参数:

```
..................................
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
             (user_id, password))
..................................
```

数据库查询操作

Python查询Mysql使用 fetchone() 方法获取单条数据, 使用fetchall() 方法获取多条数据。

- **fetchone():** 该方法获取下一个查询结果集。结果集是一个对象
- **fetchall():**接收全部的返回结果行.
- **rowcount:** 这是一个只读属性，并返回执行execute()方法后影响的行数。

实例：

查询EMPLOYEE表中salary（工资）字段大于1000的所有数据：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 查询语句
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > '%d'" % (1000)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 获取所有记录列表
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # 打印结果
      print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
             (fname, lname, age, sex, income )
except:
   print "Error: unable to fecth data"

# 关闭数据库连接
db.close()
```

以上脚本执行结果如下：

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

数据库更新操作

更新操作用于更新数据表的的数据，以下实例将 TESTDB表中的 SEX 字段全部修改为 'M'，AGE 字段递增1：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
                          WHERE SEX = '%c'" % ('M')
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

执行事务

事务机制可以确保数据一致性。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

- 原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了两个方法 commit 或 rollback。

实例：

```
# SQL删除记录语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 向数据库提交
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()
```

对于支持事务的数据库， 在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。

commit()方法游标的所有更新操作，rollback（）方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

错误处理

DB API中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常:

异常	描述
Warning	当有严重警告时触发，例如插入数据是被截断等等。必须是 StandardError 的子类。
Error	警告以外所有其他错误类。必须是 StandardError 的子类。
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。 必须是Error的子类。
DatabaseError	和数据库有关的错误发生时触发。 必须是Error的子类。
DataError	当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。 必须是DatabaseError的子类。
OperationalError	指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、 数据库名未找到、事务处理失败、内存分配错误等等操作数据库是发生的错误。 必须是DatabaseError的子类。
IntegrityError	完整性相关的错误，例如外键检查失败等。必须是DatabaseError子类。
InternalError	数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。 必须是DatabaseError子类。
ProgrammingError	程序错误，例如数据表（table）没找到或已存在、SQL语句语法错误、 参数数量错误等等。必须是DatabaseError的子类。
NotSupportedError	不支持错误，指使用了数据库不支持的函数或API等。例如在连接对象上 使用.rollback()函数，然而数据库并不支持事务或者事务已关闭。 必须是DatabaseError的子类。

Python使用SMTP发送邮件

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

python的smtplib提供了一种很方便的途径发送电子邮件。它对smtp协议进行了简单的封装。

Python创建 SMTP 对象语法如下：

```
import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

参数说明：

- host: SMTP 服务器主机。 你可以指定主机的ip地址或者域名如:w3cschool.cc，这个是可选参数。
- port: 如果你提供了 host 参数, 你需要指定 SMTP 服务使用的端口号，一般情况下SMTP端口号为25。
- local_hostname: 如果SMTP在你的本机上，你只需要指定服务器地址为 localhost 即可。

Python SMTP对象使用sendmail方法发送邮件，语法如下：

```
SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options]
```

参数说明：

- from_addr: 邮件发送者地址。
- to_addrs: 字符串列表，邮件发送地址。
- msg: 发送消息

这里要注意一下第三个参数，msg是字符串，表示邮件。我们知道邮件一般由标题，发信人，收件人，邮件内容，附件等构成，发送邮件的时候，要注意msg的格式。这个格式就是smtp协议中定义的格式。

实例

以下是一个使用Python发送邮件简单的实例：

```
#!/usr/bin/python

import smtplib

sender = 'from@fromdomain.com'
receivers = ['to@todomain.com']

message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
Subject: SMTP e-mail test

This is a test e-mail message.
"""

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

使用Python发送HTML格式的邮件

Python发送HTML格式的邮件与发送纯文本消息的邮件不同之处就是将MIMEText中_subtype设置为html。具体代码如下：

```
import smtplib
from email.mime.text import MIMEText
mailto_list=["YYY@YYY.com"]
mail_host="smtp.XXX.com"  #设置服务器
mail_user="XXX"    #用户名
mail_pass="XXXX"   #口令
mail_postfix="XXX.com"  #发件箱的后缀

def send_mail(to_list,sub,content):  #to_list：收件人；sub：主题；content：邮件内容
    me="hello"+"<"+mail_user+"@"+mail_postfix+">"   #这里的hello可以任意设置，收到信后，将按照
    msg = MIMEText(content,_subtype='html',_charset='gb2312')    #创建一个实例，这里设置为htm
    msg['Subject'] = sub    #设置主题
    msg['From'] = me
    msg['To'] = ";".join(to_list)
    try:
        s = smtplib.SMTP()
        s.connect(mail_host)  #连接smtp服务器
        s.login(mail_user,mail_pass)  #登陆服务器
        s.sendmail(me, to_list, msg.as_string())  #发送邮件
        s.close()
        return True
    except Exception, e:
        print str(e)
        return False
if __name__ == '__main__':
    if send_mail(mailto_list,"hello","<a href='http://www.cnblogs.com/xiaowuyi'>小五义</a>
        print "发送成功"
    else:
        print "发送失败"
```

或者你也可以在消息体中指定Content-type为text/html,如下实例:

```
#!/usr/bin/python

import smtplib

message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
MIME-Version: 1.0
Content-type: text/html
Subject: SMTP HTML e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>
<h1>This is headline.</h1>
"""

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

Python发送带附件的邮件

发送带附件的邮件，首先要创建MIMEMultipart()实例，然后构造附件，如果有多个附件，可依次构造，最后利用smtplib.smtp发送。

```
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import smtplib

#创建一个带附件的实例
msg = MIMEMultipart()

#构造附件1
att1 = MIMEText(open('d:\\123.rar', 'rb').read(), 'base64', 'gb2312')
att1["Content-Type"] = 'application/octet-stream'
att1["Content-Disposition"] = 'attachment; filename="123.doc"'#这里的filename可以任意写，写什
msg.attach(att1)

#构造附件2
att2 = MIMEText(open('d:\\123.txt', 'rb').read(), 'base64', 'gb2312')
att2["Content-Type"] = 'application/octet-stream'
att2["Content-Disposition"] = 'attachment; filename="123.txt"'
msg.attach(att2)

#加邮件头
msg['to'] = 'YYY@YYY.com'
msg['from'] = 'XXX@XXX.com'
msg['subject'] = 'hello world'
#发送邮件
try:
    server = smtplib.SMTP()
    server.connect('smtp.XXX.com')
    server.login('XXX','XXXXX')#XXX为用户名，XXXXX为密码
    server.sendmail(msg['from'], msg['to'],msg.as_string())
    server.quit()
    print '发送成功'
except Exception, e:
    print str(e)
```

以下实例指定了Content-type header 为 multipart/mixed，并发送/tmp/test.txt 文本文件：

```
#!/usr/bin/python

import smtplib
import base64

filename = "/tmp/test.txt"

# 读取文件内容并使用 base64 编码
fo = open(filename, "rb")
filecontent = fo.read()
encodedcontent = base64.b64encode(filecontent)  # base64

sender = 'webmaster@tutorialpoint.com'
reciever = 'amrood.admin@gmail.com'

marker = "AUNIQUEMARKER"

body ="""
This is a test email to send an attachement.
"""
# 定义头部信息
part1 = """From: From Person <me@fromdomain.net>
To: To Person <amrood.admin@gmail.com>
Subject: Sending Attachement
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=%s
--%s
""" % (marker, marker)

# 定义消息动作
part2 = """Content-Type: text/plain
Content-Transfer-Encoding:8bit

%s
--%s
""" % (body,marker)

# 定义附近部分
part3 = """Content-Type: multipart/mixed; name=\"%s\"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename=%s

%s
--%s--
""" %(filename, filename, encodedcontent, marker)
message = part1 + part2 + part3

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, reciever, message)
   print "Successfully sent email"
except Exception:
   print "Error: unable to send email"
```

Python多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
- 程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

每个线程都有他自己的一组CPU寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的CPU寄存器的状态。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

- 线程可以被抢占（中断）。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠） -- 这就是线程的退让。

开始学习Python线程

Python中使用线程有两种方式：函数或者用类来包装线程对象。

函数式：调用thread模块中的start_new_thread()函数来产生新线程。语法如下:

```
thread.start_new_thread ( function, args[, kwargs] )
```

参数说明:

- function - 线程函数。
- args - 传递给线程函数的参数,他必须是个tuple类型。
- kwargs - 可选参数。

实例：

```
#!/usr/bin/python

import thread
import time

# 为线程定义一个函数
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

# 创建两个线程
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"

while 1:
   pass
```

执行以上程序输出结果如下：

```
Thread-1: Thu Jan 22 15:42:17 2009
Thread-1: Thu Jan 22 15:42:19 2009
Thread-2: Thu Jan 22 15:42:19 2009
Thread-1: Thu Jan 22 15:42:21 2009
Thread-2: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:25 2009
Thread-2: Thu Jan 22 15:42:27 2009
Thread-2: Thu Jan 22 15:42:31 2009
Thread-2: Thu Jan 22 15:42:35 2009
```

线程的结束一般依靠线程函数的自然结束；也可以在线程函数中调用thread.exit()，他抛出SystemExit exception，达到退出线程的目的。

线程模块

Python通过两个标准库thread和threading提供对线程的支持。thread提供了低级别的、原始的线程以及一个简单的锁。

thread 模块提供的其他方法：

- threading.currentThread(): 返回当前的线程变量。
- threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。

除了使用方法外，线程模块同样提供了Thread类来处理线程，Thread类提供了以下方法:

- **run():** 用以表示线程活动的方法。
- **start():**启动线程活动。
- **join([time]):** 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- **isAlive():** 返回线程是否活动的。
- **getName():** 返回线程名。
- **setName():** 设置线程名。

使用Threading模块创建线程

使用Threading模块创建线程，直接从threading.Thread继承，然后重写**init**方法和run方法：

```
#!/usr/bin/python

import threading
import time

exitFlag = 0

class myThread (threading.Thread):   #继承父类threading.Thread
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):                   #把要执行的代码写到run函数里面 线程在创建后会直接运行run函数
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启线程
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

以上程序执行结果如下；

```
Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2
```

线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用Thread对象的Lock和Rlock可以实现简单的线程同步，这两个对象都有acquire方法和release方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到acquire和release方法之间。如下：

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。

那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。

经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

实例：

```
#!/usr/bin/python

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
       # 获得锁，成功获得锁定后返回True
       # 可选的timeout参数不填时将一直阻塞直到获得锁定
       # 否则超时后将返回False
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # 释放锁
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"
```

线程优先级队列（ Queue）

Python的Queue模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出)队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列PriorityQueue。这些队列都实现了锁原语，能够在多线程中直接使用。可以使用队列来实现线程间的同步。

Queue模块中的常用方法:

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回True,反之False
- Queue.full() 如果队列满了，返回True,反之False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout等待时间

- Queue.get_nowait() 相当Queue.get(False)
- Queue.put(item) 写入队列，timeout等待时间
- Queue.put_nowait(item) 相当Queue.put(item, False)
- Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

实例:

```
#!/usr/bin/python

import Queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print "Starting " + self.name
        process_data(self.name, self.q)
        print "Exiting " + self.name

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print "%s processing %s" % (threadName, data)
        else:
            queueLock.release()
        time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"
```

以上程序执行结果：

```
Starting Thread-1
Starting Thread-2
Starting Thread-3
Thread-1 processing One
Thread-2 processing Two
Thread-3 processing Three
Thread-1 processing Four
Thread-2 processing Five
Exiting Thread-3
Exiting Thread-1
Exiting Thread-2
Exiting Main Thread
```

python XML解析

什么是XML？

XML 指可扩展标记语言（e**X**tensible **M**arkup **L**anguage）。 你可以通过本站学习XML教程

XML 被设计用来传输和存储数据。

XML是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。

它也是元标记语言，即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。

python对XML的解析

常见的XML编程接口有DOM和SAX，这两种接口处理XML文件的方式不同，当然使用场合也不同。

python有三种方法解析XML，SAX，DOM，以及ElementTree:

1.SAX (simple API for XML)

pyhton 标准库包含SAX解析器，SAX用事件驱动模型，通过在解析XML的过程中触发一个个的事件并调用用户定义的回调函数来处理XML文件。

2.DOM(Document Object Model)

将XML数据在内存中解析成一个树，通过对树的操作来操作XML。

3.ElementTree(元素树)

ElementTree就像一个轻量级的DOM，具有方便友好的API。代码可用性好，速度快，消耗内存少。

注：因DOM需要将XML数据映射到内存中的树，一是比较慢，二是比较耗内存，而SAX流式读取XML文件，比较快，占用内存少，但需要用户实现回调函数（handler）。

本章节使用到的XML实例文件movies.xml内容如下：

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
   <type>War, Thriller</type>
   <format>DVD</format>
   <year>2003</year>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
   <type>Anime, Science Fiction</type>
   <format>DVD</format>
   <year>1989</year>
   <rating>R</rating>
   <stars>8</stars>
   <description>A schientific fiction</description>
</movie>
   <movie title="Trigun">
   <type>Anime, Action</type>
   <format>DVD</format>
   <episodes>4</episodes>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
   <type>Comedy</type>
   <format>VHS</format>
   <rating>PG</rating>
   <stars>2</stars>
   <description>Viewable boredom</description>
</movie>
</collection>
```

python使用SAX解析xml

SAX是一种基于事件驱动的API。

利用SAX解析XML文档牵涉到两个部分:解析器和事件处理器。

解析器负责读取XML文档,并向事件处理器发送事件,如元素开始跟元素结束事件;

而事件处理器则负责对事件作出相应,对传递的XML数据进行处理。

- 1、对大型文件进行处理；
- 2、只需要文件的部分内容，或者只需从文件中得到特定信息。
- 3、想建立自己的对象模型的时候。

在python中使用sax方式处理xml要先引入xml.sax中的parse函数，还有xml.sax.handler中的ContentHandler。

ContentHandler类方法介绍

characters(content)方法

调用时机：

从行开始，遇到标签之前，存在字符，content的值为这些字符串。

从一个标签，遇到下一个标签之前， 存在字符，content的值为这些字符串。

从一个标签，遇到行结束符之前，存在字符，content的值为这些字符串。

标签可以是开始标签，也可以是结束标签。

startDocument()方法

文档启动的时候调用。

endDocument()方法

解析器到达文档结尾时调用。

startElement(name, attrs)方法

遇到XML开始标签时调用，name是标签的名字，attrs是标签的属性值字典。

endElement(name)方法

遇到XML结束标签时调用。

make_parser方法

以下方法创建一个新的解析器对象并返回。

```
xml.sax.make_parser( [parser_list] )
```

参数说明:

- **parser_list** - 可选参数，解析器列表

parser方法

以下方法创建一个 SAX 解析器并解析xml文档：

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler])
```

参数说明:

- **xmlfile** - xml文件名
- **contenthandler** - 必须是一个ContentHandler的对象
- **errorhandler** - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象

parseString方法

parseString方法创建一个XML解析器并解析xml字符串：

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

参数说明:

- **xmlstring** - xml字符串
- **contenthandler** - 必须是一个ContentHandler的对象
- **errorhandler** - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象

Python 解析XML实例

```
#!/usr/bin/python

import xml.sax

class MovieHandler( xml.sax.ContentHandler ):
   def __init__(self):
      self.CurrentData = ""
      self.type = ""
      self.format = ""
      self.year = ""
      self.rating = ""
      self.stars = ""
      self.description = ""

   # 元素开始事件处理
   def startElement(self, tag, attributes):
      self.CurrentData = tag
      if tag == "movie":
         print "*****Movie*****"
         title = attributes["title"]
         print "Title:", title

   # 元素结束事件处理
   def endElement(self, tag):
      if self.CurrentData == "type":
         print "Type:", self.type
      elif self.CurrentData == "format":
         print "Format:", self.format
      elif self.CurrentData == "year":
         print "Year:", self.year
      elif self.CurrentData == "rating":
         print "Rating:", self.rating
      elif self.CurrentData == "stars":
         print "Stars:", self.stars
      elif self.CurrentData == "description":
         print "Description:", self.description
      self.CurrentData = ""

   # 内容事件处理
   def characters(self, content):
      if self.CurrentData == "type":
         self.type = content
      elif self.CurrentData == "format":
         self.format = content
      elif self.CurrentData == "year":
         self.year = content
      elif self.CurrentData == "rating":
         self.rating = content
      elif self.CurrentData == "stars":
         self.stars = content
      elif self.CurrentData == "description":
         self.description = content

if ( __name__ == "__main__"):

   # 创建一个 XMLReader
   parser = xml.sax.make_parser()
   # turn off namepsaces
   parser.setFeature(xml.sax.handler.feature_namespaces, 0)

   # 重写 ContextHandler
   Handler = MovieHandler()
   parser.setContentHandler( Handler )

   parser.parse("movies.xml")
```

以上代码执行结果如下：

```
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Year: 2003
Rating: PG
Stars: 10
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Year: 1989
Rating: R
Stars: 8
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Stars: 10
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Stars: 2
Description: Viewable boredom
```

完整的 SAX API 文档请查阅Python SAX APIs

使用xml.dom解析xml

文件对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展置标语言的标准编程接口。

一个 DOM 的解析器在解析一个 XML 文档时，一次性读取整个文档，把文档中所有元素保存在内存中的一个树结构里，之后你可以利用DOM 提供的不同的函数来读取或修改文档的内容和结构，也可以把修改过的内容写入xml文件。

python中用xml.dom.minidom来解析xml文件，实例如下：

```
#!/usr/bin/python

from xml.dom.minidom import parse
import xml.dom.minidom

# 使用minidom解析器打开 XML 文档
DOMTree = xml.dom.minidom.parse("movies.xml")
collection = DOMTree.documentElement
if collection.hasAttribute("shelf"):
   print "Root element : %s" % collection.getAttribute("shelf")

# 在集合中获取所有电影
movies = collection.getElementsByTagName("movie")

# 打印每部电影的详细信息
for movie in movies:
   print "*****Movie*****"
   if movie.hasAttribute("title"):
      print "Title: %s" % movie.getAttribute("title")

   type = movie.getElementsByTagName('type')[0]
   print "Type: %s" % type.childNodes[0].data
   format = movie.getElementsByTagName('format')[0]
   print "Format: %s" % format.childNodes[0].data
   rating = movie.getElementsByTagName('rating')[0]
   print "Rating: %s" % rating.childNodes[0].data
   description = movie.getElementsByTagName('description')[0]
   print "Description: %s" % description.childNodes[0].data
```

以上程序执行结果如下：

```
Root element : New Arrivals
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Rating: PG
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Rating: R
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Description: Viewable boredom
```

完整的 DOM API 文档请查阅Python DOM APIs。

python GUI编程(Tkinter)

python提供了多个图形开发界面的库，几个常用Python GUI库如下：

- **Tkinter**： Tkinter模块("Tk 接口")是Python的标准Tk GUI工具包的接口.Tk和Tkinter可以在大多数的Unix平台下使用,同样可以应用在Windows和Macintosh系统里.,Tk8.0的后续版本可以实现本地窗口风格,并良好地运行在绝大多数平台中。
- **wxPython**：wxPython 是一款开源软件，是 Python 语言的一套优秀的 GUI 图形库，允许 Python 程序员很方便的创建完整的、功能键全的 GUI 用户界面。
- **Jython**：Jython程序可以和Java无缝集成。除了一些标准模块，Jython使用Java的模块。Jython几乎拥有标准的Python中不依赖于C语言的全部模块。比如，Jython的用户界面将使用Swing，AWT或者SWT。Jython可以被动态或静态地编译成Java字节码。

Tkinter 编程

Tkinter 是Python的标准GUI库。Python使用Tkinter可以快速的创建GUI应用程序。

由于Tkinter是内置到python的安装包中、只要安装好Python之后就能import Tkinter库、而且IDLE也是用Tkinter编写而成、对于简单的图形界面Tkinter还是能应付自如。

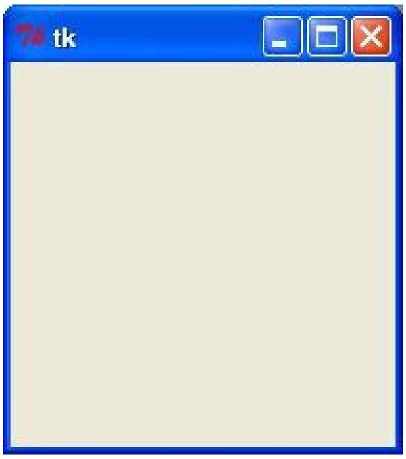
创建一个GUI程序

- 1、导入Tkinter模块
- 2、创建控件
- 3、指定这个控件的master， 即这个控件属于哪一个
- 4、告诉GM(geometry manager)有一个控件产生了。

实例:

```
#!/usr/bin/python

import Tkinter
top = Tkinter.Tk()
# 进入消息循环
top.mainloop()
```

以上代码执行结果如下图:



Tkinter 组件

Tkinter的提供各种控件，如按钮，标签和文本框，一个GUI应用程序中使用。这些控件通常被称为控件或者部件。

目前有15种Tkinter的部件。我们提出这些部件以及一个简短的介绍，在下面的表:

控件	描述
Button	按钮控件；在程序中显示按钮。
Canvas	画布控件；显示图形元素如线条或文本
Checkbutton	多选框控件；用于在程序中提供多项选择框
Entry	输入控件；用于显示简单的文本内容
Frame	框架控件；在屏幕上显示一个矩形区域，多用来作为容器
Label	标签控件；可以显示文本和位图
Listbox	列表框控件；在Listbox窗口小部件是用来显示一个字符串列表给用户
Menubutton	菜单按钮控件，由于显示菜单项。
Menu	菜单控件；显示菜单栏,下拉菜单和弹出菜单
Message	消息控件；用来显示多行文本，与label比较类似
Radiobutton	单选按钮控件；显示一个单选的按钮状态
Scale	范围控件；显示一个数值刻度，为输出限定范围的数字区间
Scrollbar	滚动条控件，当内容超过可视化区域时使用，如列表框。.
Text	文本控件；用于显示多行文本
Toplevel	容器控件；用来提供一个单独的对话框，和Frame比较类似
Spinbox	输入控件；与Entry类似，但是可以指定输入范围值
PanedWindow	PanedWindow是一个窗口布局管理的插件，可以包含一个或者多个子控件。
LabelFrame	labelframe 是一个简单的容器控件。常用与复杂的窗口布局。
tkMessageBox	用于显示你应用程序的消息框。

标准属性

标准属性也就是所有控件的共同属性，如大小，字体和颜色等等。

属性	描述
Dimension	控件大小；
Color	控件颜色；
Font	控件字体；
Anchor	锚点；
Relief	控件样式；
Bitmap	位图；
Cursor	光标；

几何管理

Tkinter控件有特定的几何状态管理方法，管理整个控件区域组织，一下是Tkinter公开的几何管理类：包、网格、位置

几何方法	描述
pack()	包装；
grid()	网格；
place()	位置；

Python2.x与3??.x版本区别

Python的3??.0版本，常被称为Python 3000，或简称Py3k。相对于Python的早期版本，这是一个较大的升级。

为了不带入过多的累赘，Python 3.0在设计的时候没有考虑向下相容。

许多针对早期Python版本设计的程式都无法在Python 3.0上正常执行。

为了照顾现有程式，Python 2.6作为一个过渡版本，基本使用了Python 2.x的语法和库，同时考虑了向Python 3.0的迁移，允许使用部分Python 3.0的语法与函数。

新的Python程式建议使用Python 3.0版本的语法。

除非执行环境无法安装Python 3.0或者程式本身使用了不支援Python 3.0的第三方库。目前不支援Python 3.0的第三方库有Twisted, py2exe, PIL等。

大多数第三方库都正在努力地相容Python 3.0版本。即使无法立即使用Python 3.0，也建议编写相容Python 3.0版本的程式，然后使用Python 2.6, Python 2.7来执行。

主要变化

Python 3.0的变化主要在以下几个方面:

print语句没有了，取而代之的是print()函数。 Python 2.6与Python 2.7部分地支持这种形式的print语法。在Python 2.6与Python 2.7里面，以下三种形式是等价的：

```
print "fish"
print ("fish") #注意print后面有个空格
print("fish") #print()不能带有任何其它参数
```

然而，Python 2.6实际已经支持新的print()语法：

```
from __future__ import print_function
print("fish", "panda", sep=', ')
```

新的str类别表示一个Unicode字串，相当于Python 2.x版本的unicode类别。而位元组序列则用类似b"abc"的语法表示，用bytes类表示，相当于Python 2.x的str类别。

现在两种类别不能再隐式地自动转换，因此在Python 3.x里面"fish"+b"panda"是错误。正确的写法是"fish"+b"panda".decode("utf-8")。 Python 2.6可以自动地将位元组序列识别为Unicode字串，方法是：

```
from __future__ import unicode_literals
print(repr("fish"))
```

除法运算符"/"在Python 3.x内总是返回浮点数。而在Python 2.6内会判断被除数与除数是否是整数。如果是整数会返回整数值，相当于整除;浮点数则返回浮点数值。

为了让Python 2.6统一返回浮点数值，可以：

```
from __future__ import division
print(3/2)
```

- 捕获异常的语法由except exc, var改为except exc as var。使用语法except (exc1, exc2) as var可以同时捕获多种类别的异常。 Python 2.6已经支援这两种语法。
- 集合(set) 的新写法：{1,2,3,4}。注意{}仍然表示空的字典(dict) 。
- 字典推导式(Dictionary comprehensions) {expr1: expr2 for k, v in d}，这个语法等价于

```
result={}
for k, v in d.items():
    result[expr1]=expr2
return result
```

集合推导式(Set Comprehensions) {expr1 for x in stuff}。这个语法等价于：

```
result = set()
for x in stuff:
    result.add(expr1)
return result
```

- 八进制数必须写成0o777，原来的形式0777不能用了；二进制必须写成0b111。新增了一个bin()函数用于将一个整数转换成二进制字串。 Python 2.6已经支援这两种语法。
- dict.keys(), dict.values??(), dict.items(), map(), filter(), range(), zip()不再返回列表，而是迭代器。
- 如果两个物件之间没有定义明确的有意义的顺序。使用<, >, <=, >=比较它们会投掷异常。比如1 < ""在Python 2.6里面会返回True，而在Python 3.0里面会投掷异常。现在cmp(), instance.**cmp**()函数已经被删除。
- 可以注释函数的参数与返回值。此特性可方便IDE对原始码进行更深入的分析。例如给参数增加类别讯息：

```
def sendMail(from_: str, to: str, title: str, body: str) -> bool:
    pass
```

- 合并int与long类型。
- 多个模块被改名（根据PEP8）：

旧的名字	新的名字
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
repr	reprlib

- StringIO模块现在被合并到新的io模组内。 new, md5, gopherlib等模块被删除。 Python 2.6已经支援新的io模组。
- httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib被合并到http包内。
- 取消了exec语句，只剩下exec()函数。 Python 2.6已经支援exec()函数。

Python IDE

本文为大家推荐几款款不错的**Python IDE**（集成开发环境），比较推荐 PyCharm，当然你可以根据自己的喜好来选择适合自己的 Python IDE。

PyCharm

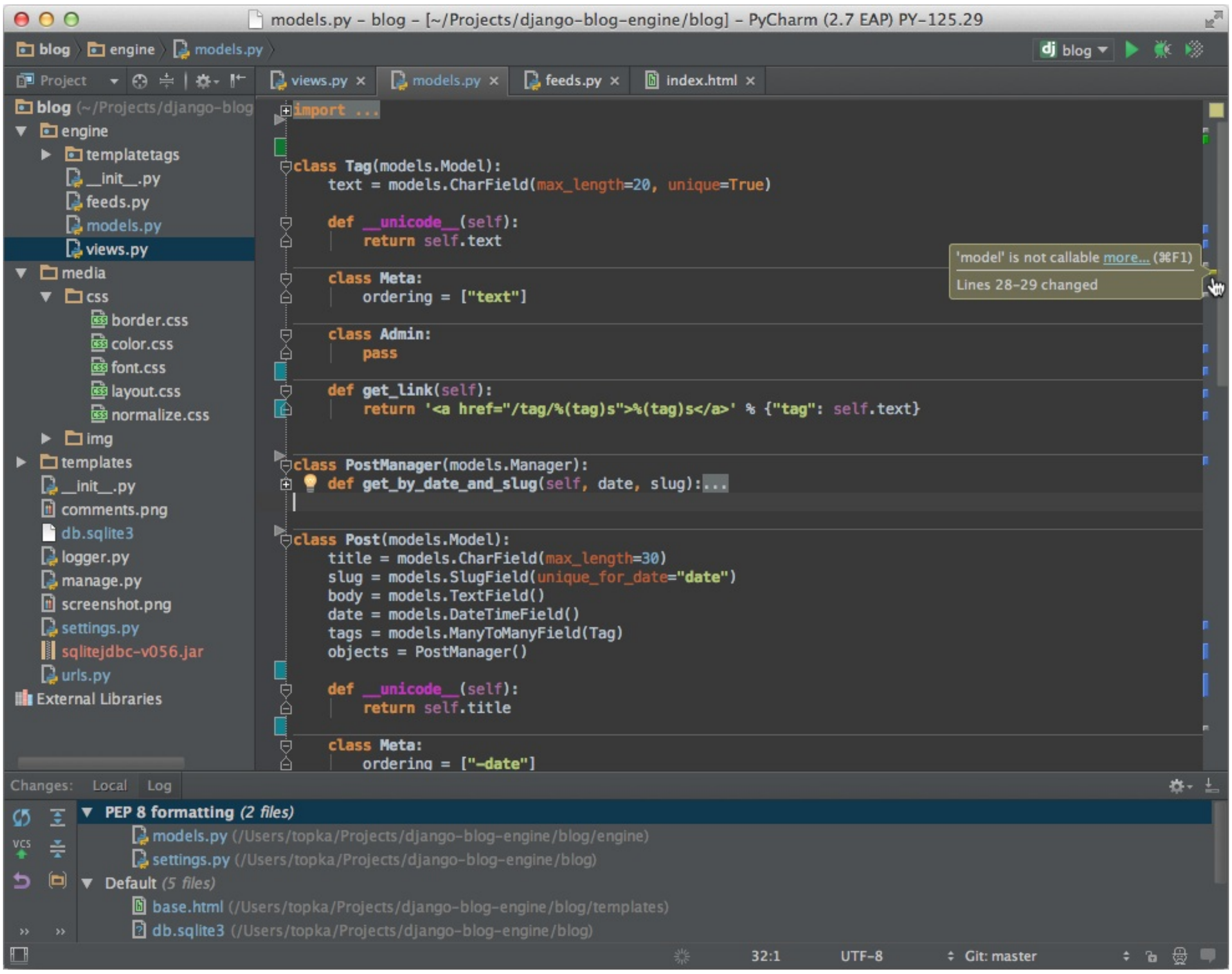
PyCharm是由JetBrains打造的一款Python IDE。

PyCharm具备一般 **Python IDE** 的功能，比如：调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。

另外，PyCharm还提供了一些很好的功能用于Django开发，同时支持Google App Engine，更酷的是，PyCharm支持IronPython。

PyCharm 官方下载地址：http://www.jetbrains.com/pycharm/download/

效果图查看：

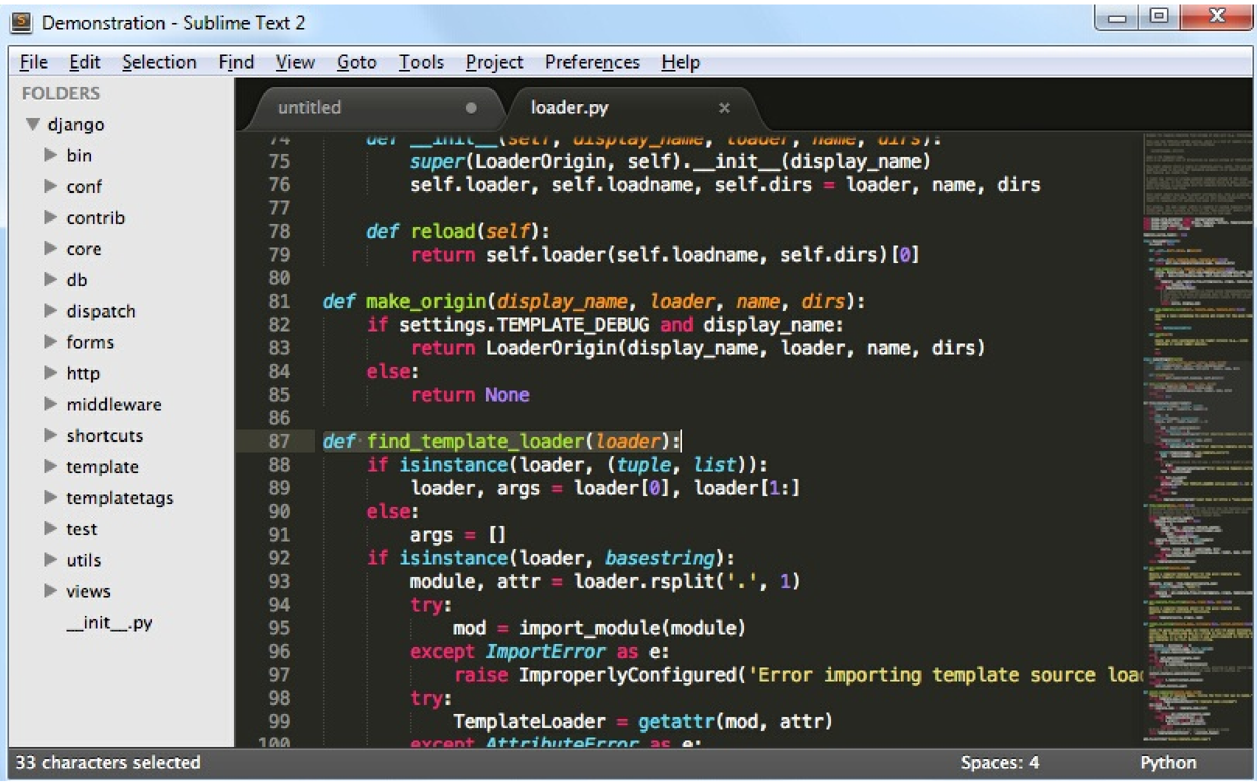


Sublinme Text 2

Sublime Text具有漂亮的用户界面和强大的功能，例如代码缩略图，Python的插件，代码段等。还可自定义键绑定，菜单和工具栏。

Sublime Text 的主要功能包括：拼写检查，书签，完整的 Python API ， Goto 功能，即时项目切换，多选择，多窗口等等。

Sublime Text 是一个跨平台的编辑器，同时支持Windows、Linux、Mac OS X等操作系统。



使用Sublinme Text 2的插件扩展功能，你可以轻松的打造一款不错的Python IDE，以下推荐几款插件（你可以找到更多）：

- CodeIntel：自动补全+成员/方法提示（强烈推荐）
- SublimeREPL：用于运行和调试一些需要交互的程序（E.G. 使用了Input()的程序）
- Bracket Highlighter：括号匹配及高亮
- SublimeLinter：代码pep8格式检查

Eclipse+Pydev

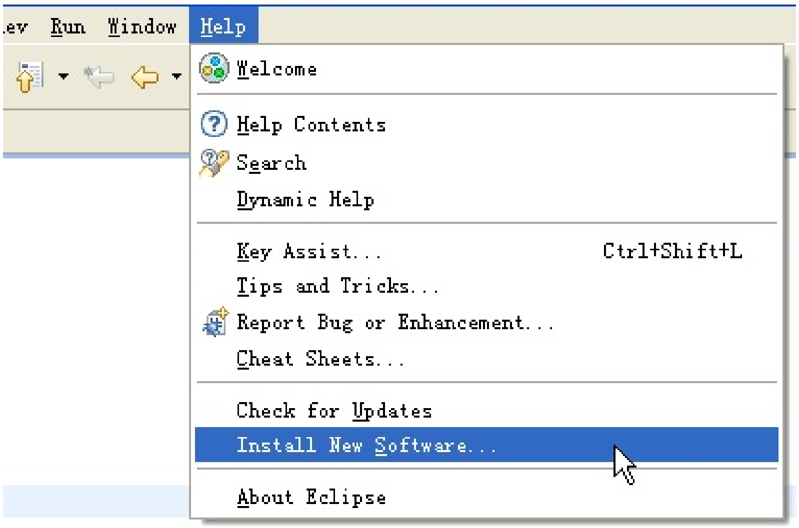
1、安装Eclipse

Eclipse可以在它的官方网站Eclipse.org找到并下载，通常我们可以选择适合自己的Eclipse版本，比如Eclipse Classic。下载完成后解压到到你想安装的目录中即可。

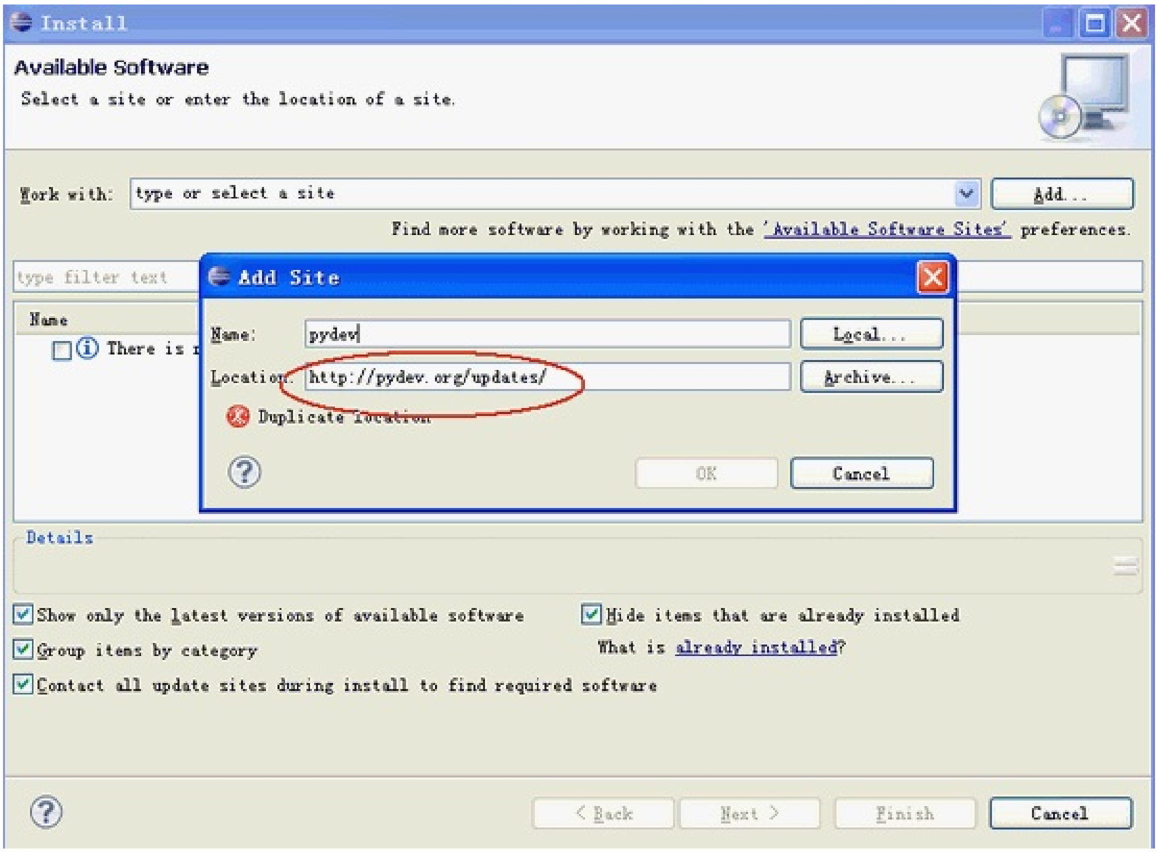
当然在执行Eclipse之前，你必须确认安装了Java运行环境,即必须安装JRE或JDK，你可以到（http://www.java.com/en/download/manual.jsp）找到JRE下载并安装。

2、安装Pydev

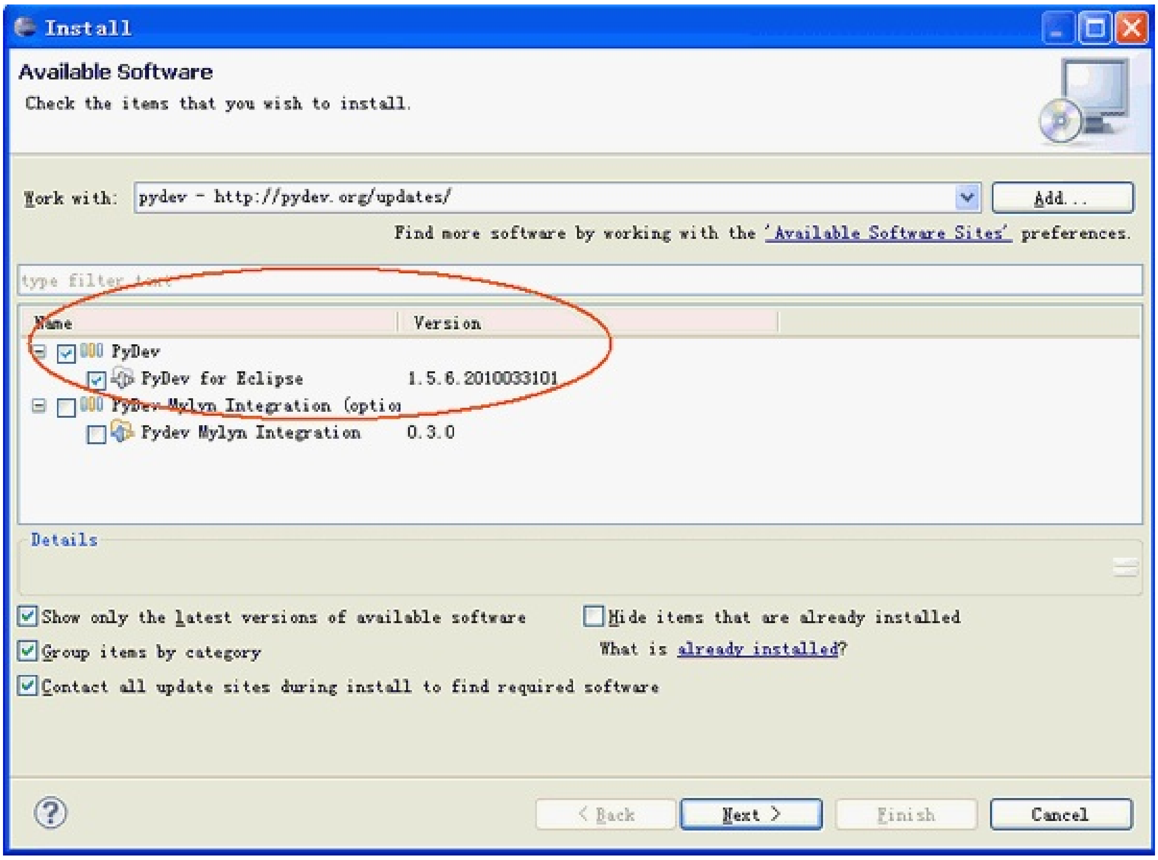
运行Eclipse之后，选择help-->Install new Software，如下图所示。



点击Add，添加pydev的安装地址：http://pydev.org/updates/，如下图所示。



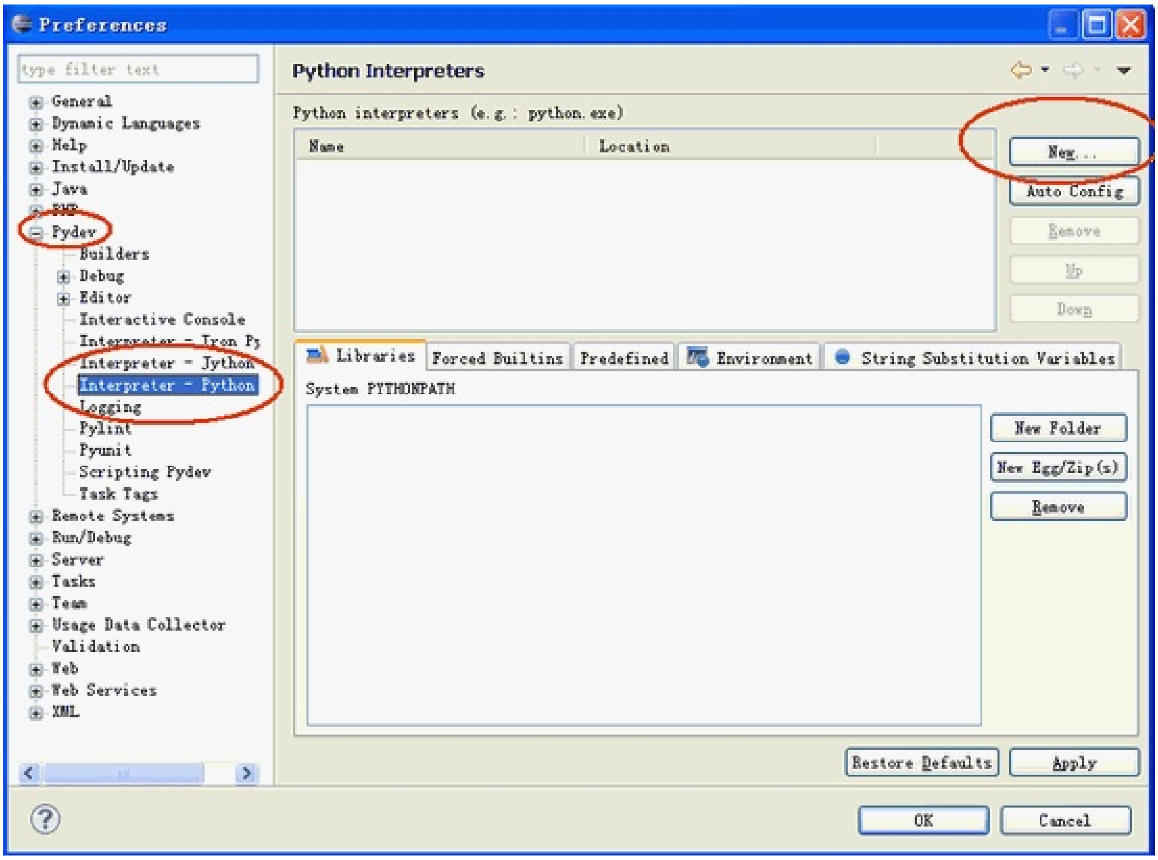
完成后点击"ok"，接着点击PyDev的"+"，展开PyDev的节点，要等一小段时间，让它从网上获取PyDev的相关套件，当完成后会多出PyDev的相关套件在子节点里，勾选它们然后按next进行安装。如下图所示。



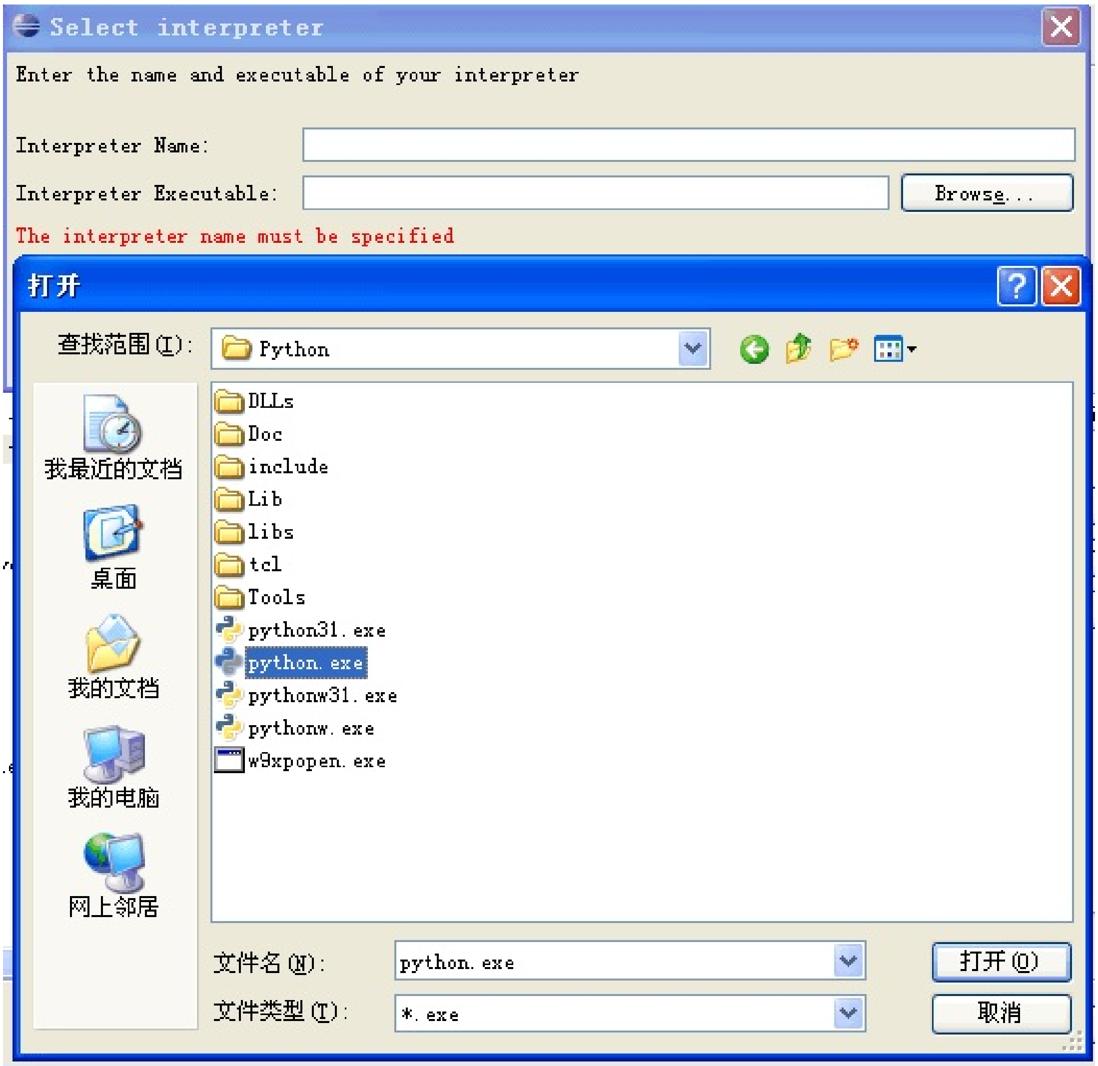
安装完成后，重启Eclipse即可

3、设置Pydev

安装完成后，还需要设置一下PyDev，选择Window -> Preferences来设置PyDev。设置Python的路径，从Pydev的Interpreter - Python页面选择New



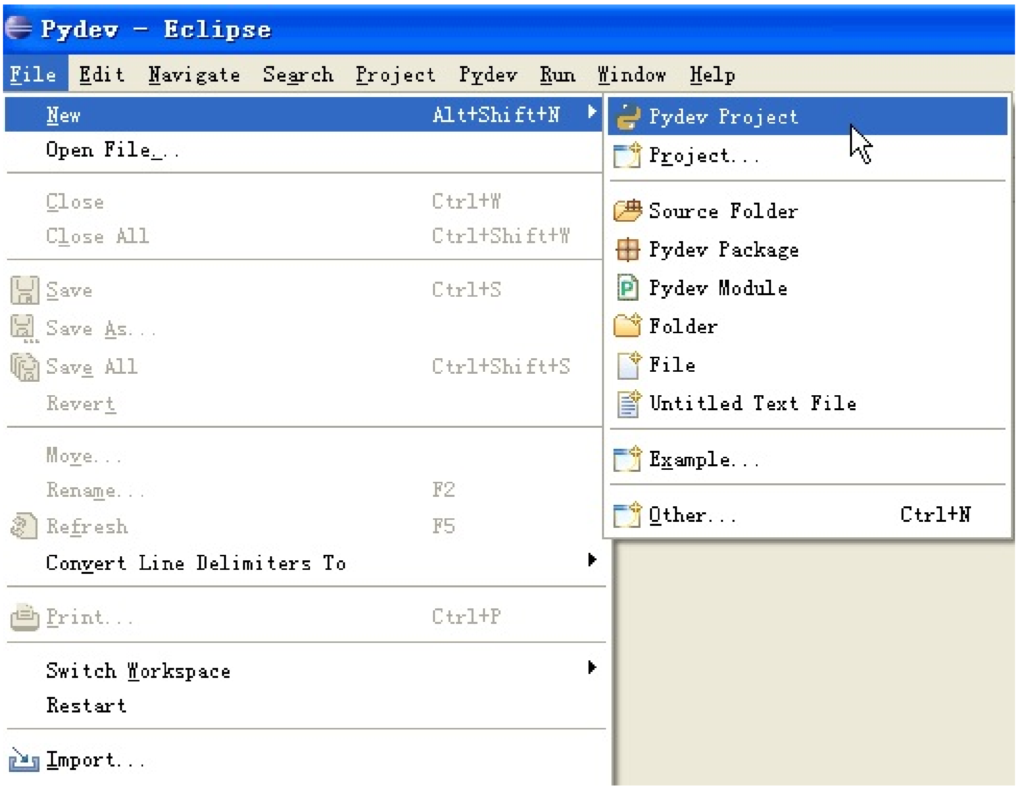
会弹出一个窗口让你选择Python的安装位置，选择你安装Python的所在位置。



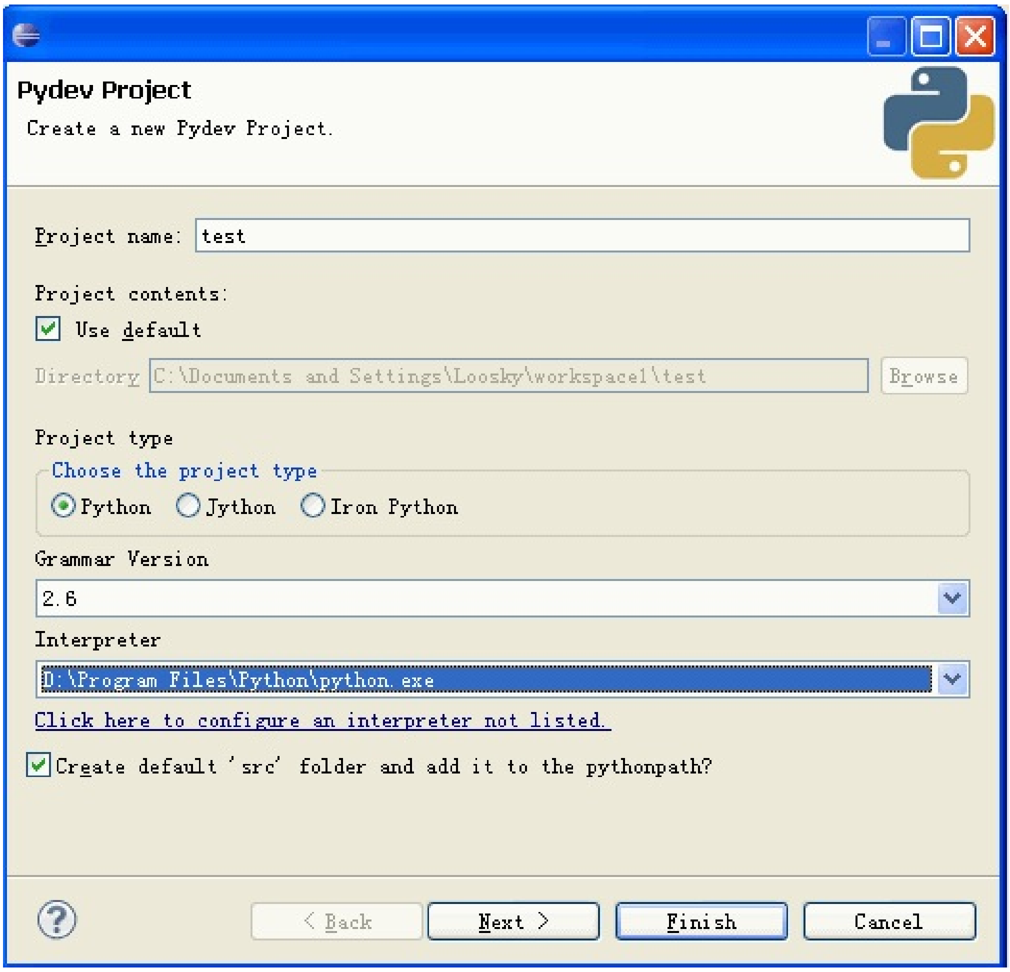
完成之后PyDev就设置完成，可以开始使用。

4、建立Python Project：

安装好Eclipse+PyDev以后，我们就可以开始使用它来开发项目了。首先要创建一个项目，选择File -> New ->Pydev Project

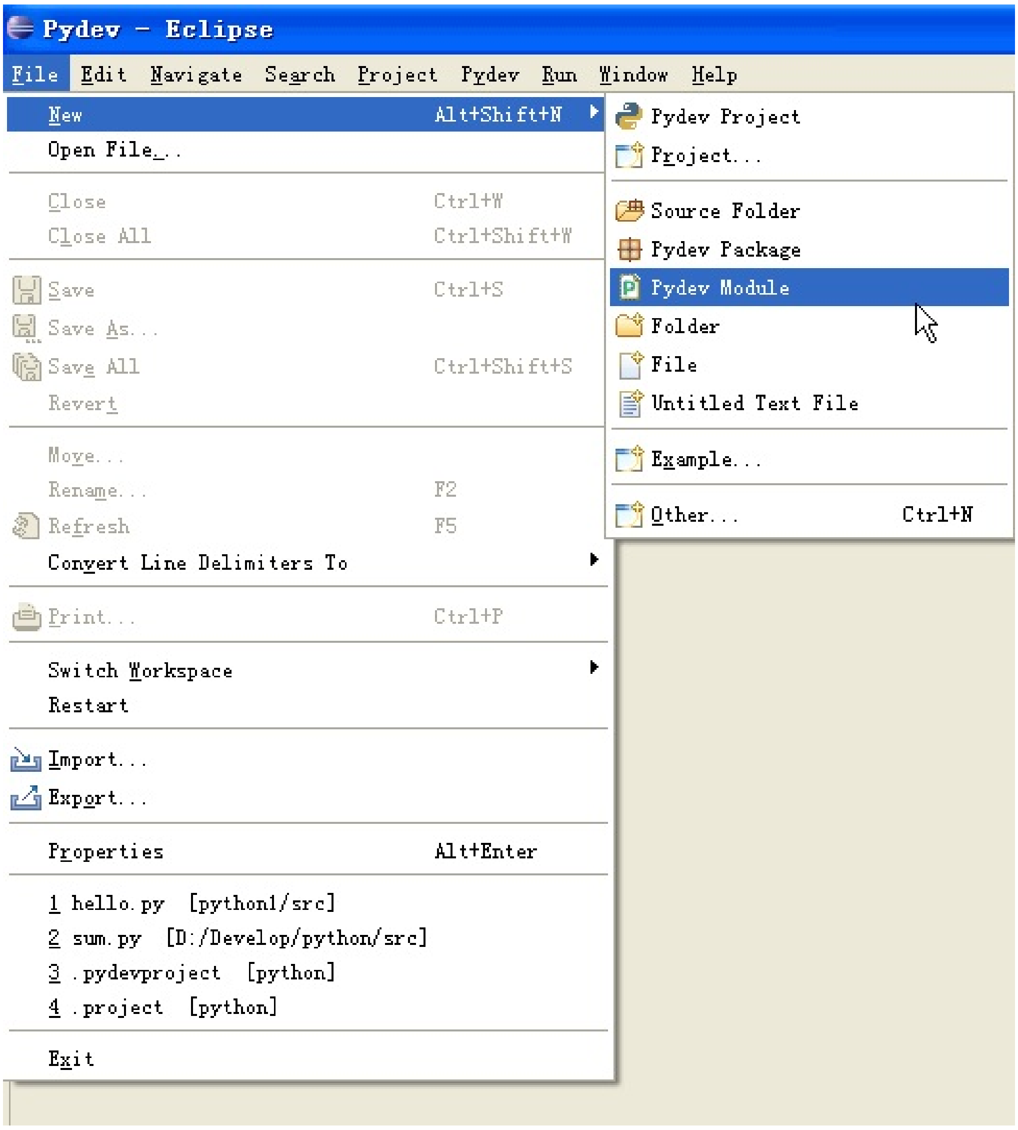


会弹出一个新窗口，填写Project Name，以及项目保存地址，然后点击next完成项目的创建。

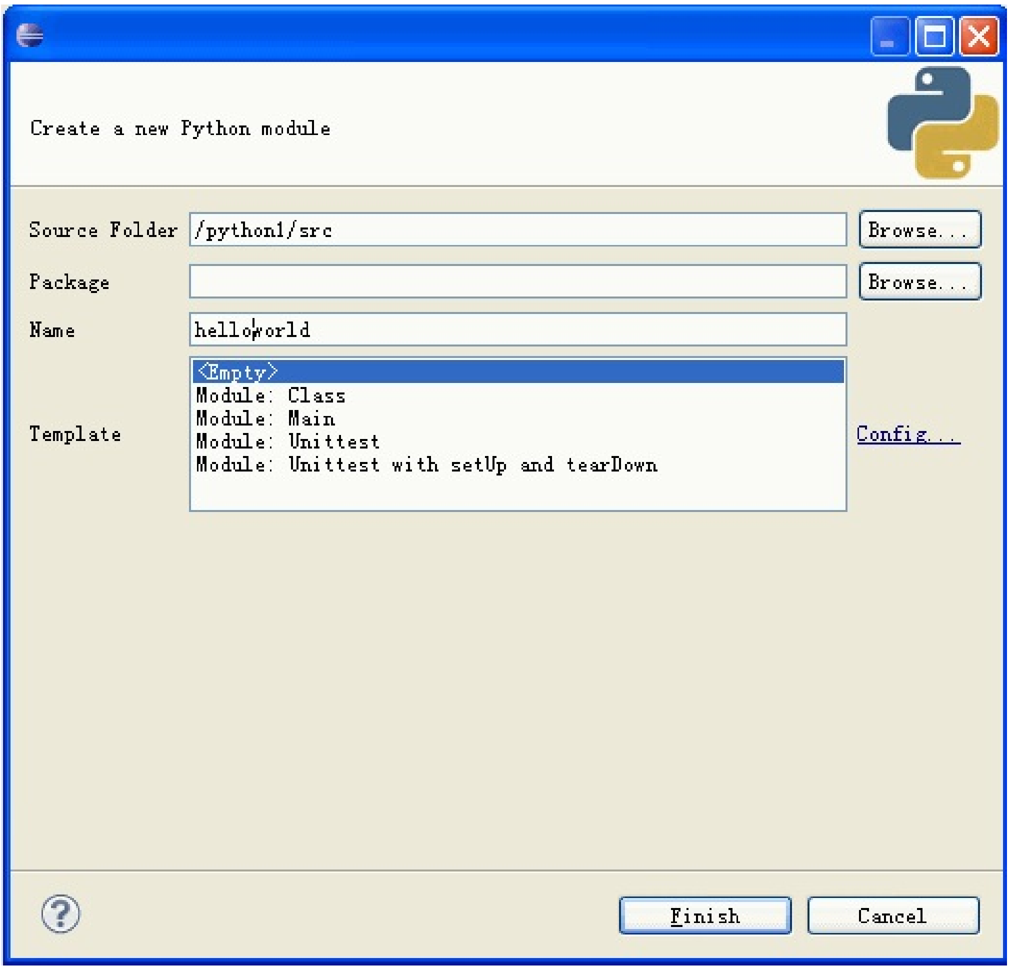


5、创建新的Pydev Module

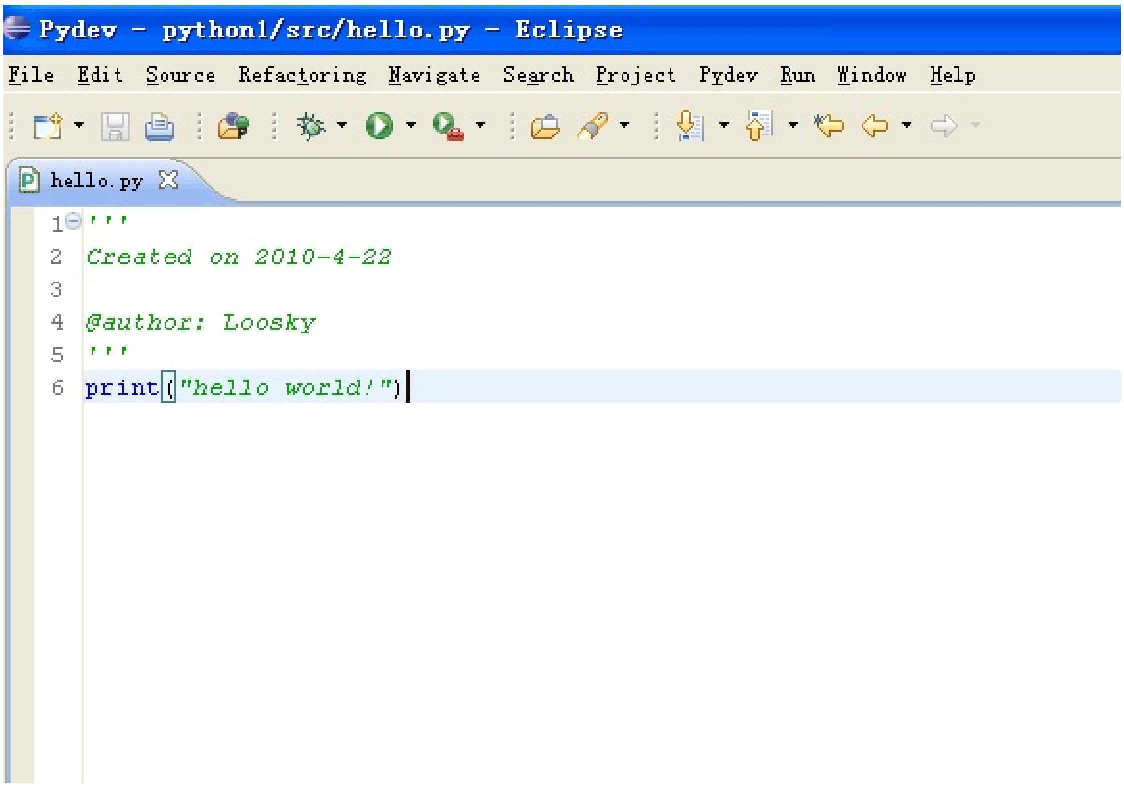
光有项目是无法执行的，接着必须创建新的Pydev Moudle，选择File -> New -> Pydev Module



在弹出的窗口中选择文件存放位置以及Moudle Name，注意Name不用加.py，它会自动帮助我们添加。然后点击Finish完成创建。

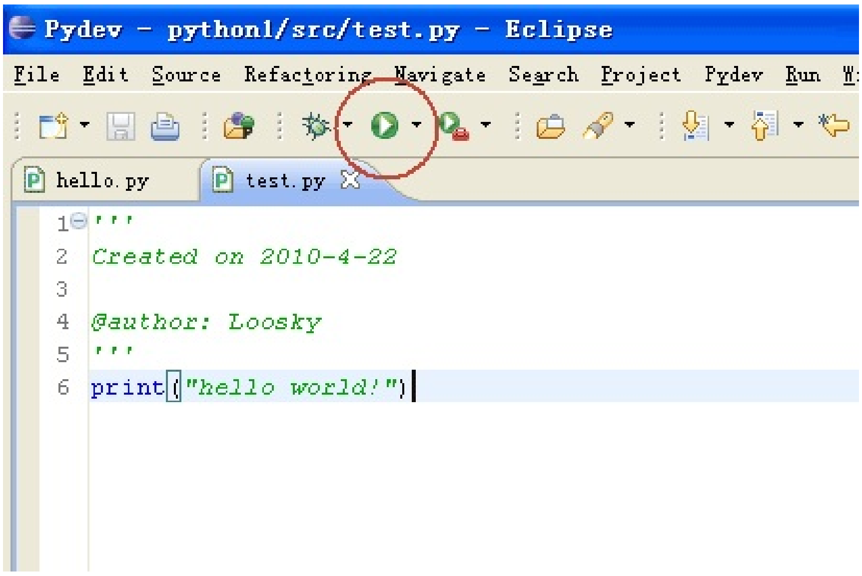


输入"hello world"的代码。



6、 执行程序

程序写完后，我们可以开始执行程序,在上方的工具栏上面找到执行的按钮。



之后会弹出一个让你选择执行方式的窗口，通常我们选择Python Run，开始执行程序。

更多 Python IDE

当然还有非常多很棒的 Python IDE，你可以自由的选择，更多 Python IDE 请参阅：http://wiki.python.org/moin/PythonEditors

Python JSON

本章节我们将为大家介绍如何使用 Python 语言来编码和解码 JSON 对象。

环境配置

在使用 Python 编码或解码 JSON 数据前，我们需要先安装 JSON 模块。本教程我们会下载 Demjson 并安装：

```
$tar xvfz demjson-1.6.tar.gz
$cd demjson-1.6
$python setup.py install
```

JSON 函数

函数	描述
encode	将 Python 对象编码成 JSON 字符串
decode	将已编码的 JSON 字符串解码为 Python 对象

encode

Python encode() 函数用于将 Python 对象编码成 JSON 字符串。

语法

```
demjson.encode(self, obj, nest_level=0)
```

实例

以下实例将数组编码为 JSON 格式数据：

```
#!/usr/bin/python
import demjson

data = [ { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5 } ]

json = demjson.encode(data)
print json
```

以上代码执行结果为：

```
[{"a":1,"b":2,"c":3,"d":4,"e":5}]
```

decode

Python 可以使用 demjson.decode() 函数解码 JSON 数据。该函数返回 Python 字段的数据类型。

语法

```
demjson.decode(self, txt)
```

实例

以下实例展示了Python 如何解码 JSON 对象：

```
#!/usr/bin/python
import demjson

json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

text = demjson.decode(json)
print  text
```

以上代码执行结果为：

```
{u'a': 1, u'c': 3, u'b': 2, u'e': 5, u'd': 4}
```

Python3 教程

Python3 基础语法

编码

默认情况下，Python 3源码文件以 UTF-8 编码，所有字符串都是 unicode 字符串。 当然你也可以为源码文件指定不同的编码：

```
# -*- coding: cp-1252 -*-
```

标识符

- 第一个字符必须是字母表中字母或下划线'_'。
- 标识符的其他的部分有字母、数字和下划线组成。
- 标识符对大小写敏感。

在Python 3中，非-ASCII 标识符也是允许的了。

python保留字

保留字即关键字，我们不能把它们用作任何标识符名称。Python的标准库提供了一个keyword module，可以输出当前版本的所有关键字：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'de
```

注释

Python中单行注释以#开头，多行注释用三个单引号（'''）或者三个双引号（"""）将注释括起来。

行与缩进

python最具特色的就是使用缩进来表示代码块。缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。

数据类型

python中数有四种类型：整数、长整数、浮点数和复数。

- 整数， 如 1
- 长整数 是比较大的整数
- 浮点数 如 1.23、3E-2
- 复数 如 1 + 2j、 1.1 + 2.2j

字符串

- python中单引号和双引号使用完全相同。
- 使用三引号('''或""")可以指定一个多行字符串。
- 转义符 '\'
- 自然字符串， 通过在字符串前加r或R。 如 r"this is a line with \n" 则\n会显示，并不是换行。
- python允许处理unicode字符串，加前缀u或U， 如 u"this is an unicode string"。
- 字符串是不可变的。
- 按字面意义级联字符串，如"this " "is " "string"会被自动转换为this is string。

Python3 基本数据类型

Python中的变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

在Python中，变量就是变量，它没有类型，我们所说的"类型"是变量所指的内存中对象的类型。

Python 3中有六个标准的数据类型：

- Numbers（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Sets（集合）
- Dictionaries（字典）

Numbers（数字）

Python 3支持int、float、bool、complex（复数）。

数值类型的赋值和计算都是很直观的，就像大多数语言一样。内置的type()函数可以用来查询变量所指的对象类型。

```
>>> a, b, c, d = 20, 5.5, True, 4+3j
>>> print(type(a), type(b), type(c), type(d))
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
```

数值运算：

```
>>> 5 + 4  # 加法
9
>>> 4.3 - 2 # 减法
2.3
>>> 3 * 7  # 乘法
21
>>> 2 / 4  # 除法，得到一个浮点数
0.5
>>> 2 // 4 # 除法，得到一个整数
0
>>> 17 % 3 # 取余
2
>>> 2 ** 5 # 乘方
32
```

注意：

- 1、Python可以同时为多个变量赋值，如a, b = 1, 2。
- 2、一个变量可以通过赋值指向不同类型的对象。
- 3、数值的除法（/）总是返回一个浮点数，要获取整数使用//操作符。
- 4、在混合计算时，Pyhton会把整型转换成为浮点数。

String（字符串）

Python中的字符串str用单引号(' ')或双引号(" ")括起来，同时使用反斜杠()转义特殊字符。

```
>>> s = 'Yes,he doesn\'t'
>>> print(s, type(s), len(s))
Yes,he doesn't <class 'str'> 14
```

如果你不想让反斜杠发生转义，可以在字符串前面添加一个r，表示原始字符串：

```
>>> print('C:\some\name')
C:\some
ame
>>> print(r'C:\some\name')
C:\some\name
```

另外，反斜杠可以作为续行符，表示下一行是上一行的延续。还可以使用"""..."""或者'''...'''跨越多行。

字符串可以使用 + 运算符串连接在一起，或者用 * 运算符重复：

```
>>> print('str'+'ing', 'my'*3)
string mymymy
```

Python中的字符串有两种索引方式，第一种是从左往右，从0开始依次增加；第二种是从右往左，从-1开始依次减少。

注意，没有单独的字符类型，一个字符就是长度为1的字符串。

```
>>> word = 'Python'
>>> print(word[0], word[5])
P n
>>> print(word[-1], word[-6])
n P
```

还可以对字符串进行切片，获取一段子串。用冒号分隔两个索引，形式为变量[头下标:尾下标]。

截取的范围是前闭后开的，并且两个索引都可以省略：

```
>>> word = 'ilovepython'
>>> word[1:5]
'love'
>>> word[:]
'ilovepython'
>>> word[5:]
'python'
>>> word[-10:-6]
'love'
```

与C字符串不同的是，Python字符串不能被改变。向一个索引位置赋值，比如word[0] = 'm'会导致错误。

注意：

- 1、反斜杠可以用来转义，使用r可以让反斜杠不发生转义。
- 2、字符串可以用+运算符连接在一起，用*运算符重复。
- 3、Python中的字符串有两种索引方式，从左往右以0开始，从右往左以-1开始。
- 4、Python中的字符串不能改变。

List（列表）

List（列表） 是 Python 中使用最频繁的数据类型。

列表是写在方括号之间、用逗号分隔开的元素列表。列表中元素的类型可以不相同：

```
>>> a = ['him', 25, 100, 'her']
>>> print(a)
['him', 25, 100, 'her']
```

和字符串一样，列表同样可以被索引和切片，列表被切片后返回一个包含所需元素的新列表。详细的在这里就不赘述了。

列表还支持串联操作，使用+操作符：

```
>>> a = [1, 2, 3, 4, 5]
>>> a + [6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

与Python字符串不一样的是，列表中的元素是可以改变的：

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> a[0] = 9
>>> a[2:5] = [13, 14, 15]
>>> a
[9, 2, 13, 14, 15, 6]
>>> a[2:5] = []   # 删除
>>> a
[9, 2, 6]
```

List内置了有很多方法，例如append()、pop()等等，这在后面会讲到。

注意：

- 1、List写在方括号之间，元素用逗号隔开。
- 2、和字符串一样，list可以被索引和切片。
- 3、List可以使用+操作符进行拼接。
- 4、List中的元素是可以改变的。

Tuple（元组）

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号里，元素之间用逗号隔开。

元组中的元素类型也可以不相同：

```
>>> a = (1991, 2014, 'physics', 'math')
>>> print(a, type(a), len(a))
(1991, 2014, 'physics', 'math') <class 'tuple'> 4
```

元组与字符串类似，可以被索引且下标索引从0开始，也可以进行截取/切片（看上面，这里不再赘述）。

其实，可以把字符串看作一种特殊的元组。

```
>>> tup = (1, 2, 3, 4, 5, 6)
>>> print(tup[0], tup[1:5])
1 (2, 3, 4, 5)
>>> tup[0] = 11  # 修改元组元素的操作是非法的
```

虽然tuple的元素不可改变，但它可以包含可变的对象，比如list列表。

构造包含0个或1个元素的tuple是个特殊的问题，所以有一些额外的语法规则：

```
tup1 = () # 空元组
tup2 = (20,) # 一个元素，需要在元素后添加逗号
```

另外，元组也支持用+操作符：

```
>>> tup1, tup2 = (1, 2, 3), (4, 5, 6)
>>> print(tup1+tup2)
(1, 2, 3, 4, 5, 6)
```

string、list和tuple都属于sequence（序列）。

注意：

- 1、与字符串一样，元组的元素不能修改。
- 2、元组也可以被索引和切片，方法一样。
- 3、注意构造包含0或1个元素的元组的特殊语法规则。
- 4、元组也可以使用+操作符进行拼接。

Sets（集合）

集合（set）是一个无序不重复元素的集。

基本功能是进行成员关系测试和消除重复元素。

可以使用大括号 或者 set()函数创建set集合，注意：创建一个空集合必须用 set() 而不是 { }，因为{ }是用来创建一个空字典。

```
>>> student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'}
>>> print(student)   # 重复的元素被自动去掉
{'Jim', 'Jack', 'Mary', 'Tom', 'Rose'}
>>> 'Rose' in student  # membership testing（成员测试）
True
>>> # set可以进行集合运算
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'b', 'c', 'd', 'r'}
>>> a - b     # a和b的差集
{'b', 'd', 'r'}
>>> a | b     # a和b的并集
{'l', 'm', 'a', 'b', 'c', 'd', 'z', 'r'}
>>> a & b     # a和b的交集
{'a', 'c'}
>>> a ^ b     # a和b中不同时存在的元素
{'l', 'm', 'b', 'd', 'z', 'r'}
```

Dictionaries（字典）

字典（dictionary）是Python中另一个非常有用的内置数据类型。

字典是一种映射类型（mapping type），它是一个无序的键：值对集合。

关键字必须使用不可变类型，也就是说list和包含可变类型的tuple不能做关键字。

在同一个字典中，关键字还必须互不相同。

```
>>> dic = {}  # 创建空字典
>>> tel = {'Jack':1557, 'Tom':1320, 'Rose':1886}
>>> tel
{'Tom': 1320, 'Jack': 1557, 'Rose': 1886}
>>> tel['Jack']   # 主要的操作：通过key查询
1557
>>> del tel['Rose']  # 删除一个键值对
>>> tel['Mary'] = 4127  # 添加一个键值对
>>> tel
{'Tom': 1320, 'Jack': 1557, 'Mary': 4127}
>>> list(tel.keys())  # 返回所有key组成的list
['Tom', 'Jack', 'Mary']
>>> sorted(tel.keys()) # 按key排序
['Jack', 'Mary', 'Tom']
>>> 'Tom' in tel       # 成员测试
True
>>> 'Mary' not in tel  # 成员测试
False
```

构造函数 dict() 直接从键值对sequence中构建字典，当然也可以进行推导，如下：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'jack': 4098, 'sape': 4139, 'guido': 4127}

>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}

>>> dict(sape=4139, guido=4127, jack=4098)
{'jack': 4098, 'sape': 4139, 'guido': 4127}
```

另外，字典类型也有一些内置的函数，例如clear()、keys()、values()等。

注意：

- 1、字典是一种映射类型，它的元素是键值对。
- 2、字典的关键字必须为不可变类型，且不能重复。
- 3、创建空字典使用{ }。

Python解释器

Linux/Unix的系统上，Python解释器通常被安装在 /usr/local/bin/python3.4 这样的有效路径（目录）里。

我们可以将路径 /usr/local/bin 添加到您的Linux/Unix操作系统的环境变量中，这样您就可以通过 shell 终端输入下面的命令来启动 Python 。

```
python3.4
```

在Window系统下你可以通过以下命令来设置Python的环境变量，假设你的Python安装在 C:\Python34 下:

```
set path=%path%;C:\python34
```

交互式编程

我们可以在命令提示符中输入"Python"命令来启动Python解释器：

```
python
```

执行以上命令后，出现如下窗口信息：

```
$ python3.4
Python 3.4 (default, Mar 16 2014, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 python 提示符中输入以下语句，然后按回车键查看运行效果：

```
print ("Hello, Python!");
```

以上命令执行结果如下：

```
Hello, Python!
```

当键入一个多行结构时，续行是必须的。我们可以看下如下 if 语句：

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

脚本式编程

将如下代码拷贝至hello.py文件中：

```
print ("Hello, Python!");
```

通过以下命令执行该脚本：

```
python hello.py
```

输出结果为：

```
Hello, Python!
```

在Linux/Unix系统中，你可以在脚本顶部添加以下命令让Python脚本可以像SHELL脚本一样可直接执行：

```
#! /usr/bin/env python3.4
```

然后修改脚本权限，使其有执行权限，命令如下：

```
$ chmod +x hello.py
```

执行以下命令：

```
./hello.py
```

输出结果为：

```
Hello, Python!
```

有关Python基础语法部分请参阅：Python基础语法

Python 注释

确保对模块, 函数, 方法和行内注释使用正确的风格

Python中的注释有单行注释和多行注释：

Python中单行注释以#开头，例如：：

```
# 这是一个注释
print("Hello, World!")
```

多行注释用三个单引号（'''）或者三个双引号（"""）将注释括起来，例如:

1、单引号（'''）

```
#!/usr/bin/python3
'''
这是多行注释，用三个单引号
这是多行注释，用三个单引号
这是多行注释，用三个单引号
'''
print("Hello, World!")
```

2、双引号（'''）

```
#!/usr/bin/python3
"""
这是多行注释，用三个单引号
这是多行注释，用三个单引号
这是多行注释，用三个单引号
"""
print("Hello, World!")
```

Python 数字运算

Python 解释器可以作为一个简单的计算器：您可以在解释器里输入一个表达式，它将输出表达式的值。

表达式的语法很直白： +, -, * 和/ 和在许多其它语言（如Pascal或C）里一样；括号可以用来为运算分组。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # 总是返回一个浮点数
1.6
```

注意：在不同的机器上浮点运算的结果可能会不一样。之后我们会介绍有关控制浮点运算输出结果的内容。

在整数除法中，除法（/）总是返回一个浮点数，如果只想得到整数的结果，丢弃可能的分数部分，可以使用运算符 // ：

```
>>> 17 / 3  # 整数除法返回浮点型
5.666666666666667
>>>
>>> 17 // 3  # 整数除法返回向下取整后的结果
5
>>> 17 % 3  # %操作符返回除法的余数
2
>>> 5 * 3 + 2
17
```

等号（'='）用于给变量赋值。赋值之后，除了下一个提示符，解释器不会显示任何结果。

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Python 可以使用**操作来进行幂运算：

```
>>> 5 ** 2  # 5 的平方
25
>>> 2 ** 7  # 2的7次方
128
```

变量在使用前必须先"定义"（即赋予变量一个值），否则会出现错误：

```
>>> # 尝试访问一个未定义的变量
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

浮点数得到完全的支持；不同类型的数混合运算时会将整数转换为浮点数：

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

在交互模式中，最后被输出的表达式结果被赋值给变量 _ 。这能使您在把Python作为一个桌面计算器使用时使后续计算更方便，例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

此处， _ 变量应被用户视为只读变量。不要显式地给它赋值——这样您将会创建一个具有相同名称的独立的本地变量，并且屏蔽了这个内置变量的功能。

Python 字符串

除了数字，Python也能操作字符串。字符串有几种表达方式，可以使用单引号或双引号括起来：

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Python中使用反斜杠转义引号和其它特殊字符来准确地表示。

如果字符串包含有单引号但不含双引号，则字符串会用双引号括起来，否则用单引号括起来。对于这样的输入字符串，print() 函数会产生更易读的输出。

跨行的字面字符串可用以下几种方法表示。使用续行符，即在每行最后一个字符后使用反斜线来说明下一行是上一行逻辑上的延续：

以下使用 \n 来添加新行：

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n 意味着新行
>>> s  # 不使用 print(), \n 包含在输出中
'First line.\nSecond line.'
>>> print(s)  # 使用 print(), \n 输出一个新行
First line.
Second line.
```

以下使用 反斜线（\） 来续行：

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant."

print(hello)
```

注意，其中的换行符仍然要使用 \n 表示——反斜杠后的换行符被丢弃了。以上例子将如下输出：

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

或者，字符串可以被 """ （三个双引号）或者 ''' （三个单引号）括起来。使用三引号时，换行符不需要转义，它们会包含在字符串中。以下的例子使用了一个转义符，避免在最开始产生一个不需要的空行。

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

其输出如下：

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

如果我们使用"原始"字符串，那么 \n 不会被转换成换行，行末的的反斜杠，以及源码中的换行符，都将作为数据包含在字符串内。例如：

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print(hello)
```

将会输出：

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

字符串可以使用 + 运算符串连接在一起，或者用 * 运算符重复：

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

两个紧邻的字面字符串将自动被串连；上例的第一行也可以写成 word = 'Help' 'A' ；这样的操作只在两个字面值间有效，不能随意用于字符串表达式中：

```
>>> 'str' 'ing'                   #  <-  这样操作正确
'string'
>>> 'str'.strip() + 'ing'   #  <-  这样操作正确
'string'
>>> 'str'.strip() 'ing'     #  <-  这样操作错误
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                  ^
SyntaxError: invalid syntax
```

字符串可以被索引；就像 C 语言一样，字符串的第一个字符的索引为 0。没有单独的字符类型；一个字符就是长度为一的字符串。就像Icon编程语言一样，子字符串可以使用分切符来指定：用冒号分隔的两个索引。

```
>>> word[4]
'A'
>>> word[0:2]
'Hl'
>>> word[2:4]
'ep'
```

默认的分切索引很有用：默认的第一个索引为零，第二个索引默认为字符串可以被分切的长度。

```
>>> word[:2]    # 前两个字符
'He'
>>> word[2:]    # 除了前两个字符之外，其后的所有字符
'lpA'
```

不同于C字符串的是，Python字符串不能被改变。向一个索引位置赋值会导致错误：

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object does not support slice assignment
```

然而，用组合内容的方法来创建新的字符串是简单高效的：

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
在分切操作字符串时，有一个很有用的规律： s[:i] + s[i:] 等于 s.

>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

对于有偏差的分切索引的处理方式也很优雅：一个过大的索引将被字符串的大小取代，上限值小于下限值将返回一个空字符串。

```
>>> word[1:100]
'elpA'
>>> word[10:]

>>> word[2:1]
```

在索引中可以使用负数，这将会从右往左计数。例如：

```
>>> word[-1]     # 最后一个字符
'A'
>>> word[-2]     # 倒数第二个字符
'p'
>>> word[-2:]    # 最后两个字符
'pA'
>>> word[:-2]    # 除了最后两个字符之外，其前面的所有字符
'Hel'
但要注意， -0 和 0 完全一样，所以 -0 不会从右开始计数！

>>> word[-0]     # (既然 -0 等于 0)
'H'
```

超出范围的负数索引会被截去多余部分，但不要尝试在一个单元素索引（非分切索引）里使用：

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # 错误
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

有一个方法可以让您记住分切索引的工作方式，想像索引是指向字符之间，第一个字符左边的数字是 0。接着，有n个字符的字符串最后一个字符的右边是索引n，例如：

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

第一行的数字 0...5 给出了字符串中索引的位置；第二行给出了相应的负数索引。分切部分从 i 到 j 分别由在边缘被标记为 i 和 j 的全部字符组成。

对于非负数分切部分，如果索引都在有效范围内，分切部分的长度就是索引的差值。例如，word[1:3] 的长度是2。

内置的函数 len() 用于返回一个字符串的长度：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Python 列表

Python囊括了大量的复合数据类型，用于组织其它数值。最有用的是列表，即写在方括号之间、用逗号分隔开的数值列表。列表内的项目不必全是相同的类型。

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

像字符串一样，列表可以被索引和切片：

```
<pre>
>>> squares[0]  # 索引返回的指定项
1
>>> squares[-1]
25
>>> squares[-3:]  # 切割列表并返回新的列表
[9, 16, 25]
```

所有的分切操作返回一个包含有所需元素的新列表。如下例中，分切将返回列表 squares 的一个拷贝：

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

列表还支持拼接操作：

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Python 字符串是固定的，列表可以改变其中的元素：

```
>>> cubes = [1, 8, 27, 65, 125]
>>> 4 ** 3
64
>>> cubes[3] = 64  # 修改列表值
>>> cubes
[1, 8, 27, 64, 125]
```

您也可以通过使用append()方法在列表的末尾添加新项：

```
>>> cubes.append(216)  # cube列表中添加新值
>>> cubes.append(7 ** 3)  #  cube列表中添加第七个值
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

你也可以修改指定区间的列表值：

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # 替换一些值
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # 移除值
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # 清楚列表
>>> letters[:] = []
>>> letters
[]
```

内置函数 len() 用于统计列表：

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

也可以使用嵌套列表（在列表里创建其它列表），例如：

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Python 编程第一步

现在，我们能使用Python完成比 2+2 更复杂的工作。在下例里，我们能写出一个初步的斐波纳契数列如下：

```
>>> # Fibonacci series: 斐波纳契数列
... # 两个元素的总和确定了下一个数
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

这个例子介绍了几个新特征。

- 第一行包含了一个复合赋值：变量 a 和 b 同时得到新值 0 和 1。最后一行再次使用了同样的方法，可以看到，右边的表达式会在赋值变动之前执行。右边表达式的执行顺序是从左往右的。

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

关键字end可以被用于防止输出新的一行，或者在输出的末尾添加不同的字符：

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Python 条件控制

if 语句

Python中if语句的一般形式如下所示：

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

如果 "condition_1" 为 True 将执行 "statement_block_1" 块语句，如果 "condition_1" 为 False，将判断 "condition_2"，如果"condition_2" 为 True 将执行 "statement_block_2" 块语句，如果 "condition_2" 为False，将执行"statement_block_3"块语句。

Python中用elif代替了else if，所以if语句的关键字为：if – elif – else。

注意：

- 1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。
- 2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
- 3、在Python中没有switch – case语句。

实例

以下实例演示了狗的年龄计算判断：

```
age = int(input("Age of the dog: "))
print()
if age < 0:
    print("This can hardly be true!")
elif age == 1:
    print("about 14 human years")
elif age == 2:
    print("about 22 human years")
elif age > 2:
    human = 22 + (age -2)*5
    print("Human years: ", human)

###
input('press Return>')
```

将以上脚本保存在dog.py文件中，并执行该脚本：

```
python dog.py
Age of the dog: 1

about 14 human years
```

以下为if中常用的操作运算符:

操作符	描述
`<`	小于
`<=`	小于或等于
`>`	大于
`>=`	大于或等于
`==`	等于，比较对象是否相等
`!=`	不等于

实例

```
# 程序演示了 == 操作符
# 使用数字
print(5 == 6)
# 使用变量
x = 5
y = 8
print(x == y)
```

以上实例输出结果：

```
False
False
```

high_low.py文件：

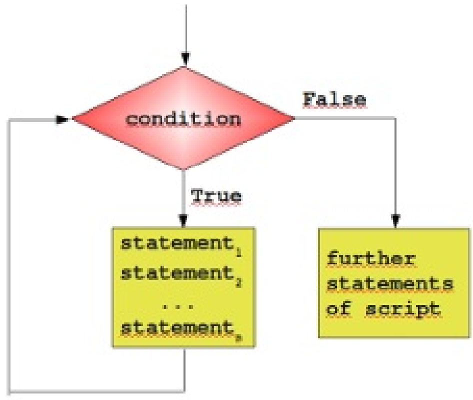
```
#!/usr/bin/python3
# 该实例演示了数字猜谜游戏
number = 7
guess = -1
print("Guess the number!")
while guess != number:
    guess = int(input("Is it... "))

    if guess == number:
        print("Hooray! You guessed it right!")
    elif guess < number:
        print("It's bigger...")
    elif guess > number:
        print("It's not so big.")
```

Python 循环

本章节将为大家介绍Python循环语句的使用。

Python中的循环语句有 for 和 while。

Python循环语句的控制结构图如下所示：



while 循环

Python中while语句的一般形式：

```
while 判断条件：
    statements
```

同样需要注意冒号和缩进。另外，在Python中没有do..while循环。

以下实例使用了 while 来计算 1 到 100 的总和：

```
#!/usr/bin/env python3

n = 100

sum = 0
counter = 1
while counter <= n:
    sum = sum + counter
    counter += 1

print("Sum of 1 until %d: %d" % (n,sum))
```

执行结果如下：

```
Sum of 1 until 100: 5050
```

for语句

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

for循环的一般格式如下：

```
for <variable> in <sequence>:
    <statements>
else:
    <statements>
```

Python loop循环实例：

```
>>> languages = ["C", "C++", "Perl", "Python"]
>>> for x in languages:
...     print x
...
C
C++
Perl
Python
>>>
```

以下实例for实例中使用了 break语句，break语句用于跳出当前循环体：

```
#!/usr/bin/env python3
edibles = ["ham", "spam","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

执行脚本后，在循环到"spam"时会跳出循环体：

```
Great, delicious ham
No more spam please!
Finally, I finished stuffing myself
```

range()函数

如果你需要遍历数字序列，可以使用内置range()函数。它会生成数列，例如:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

你也可以使用range指定区间的值：

```
>>> for i in range(5,9) :
    print(i)

5
6
7
8
>>>
```

也可以使range以指定数字开始并指定不同的增量(甚至可以是负数;有时这也叫做'步长'):

```
>>> for i in range(0, 10, 3) :
    print(i)

0
3
6
9
>>>
```

负数：

```
>>> for i in range(-10, -100, -30) :
    print(i)

-10
-40
-70
>>>
```

您可以结合range()和len()函数以遍历一个序列的索引,如下所示:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

还可以使用range()函数来创建一个列表：

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>>
```

break和continue语句及循环中的else子句

break语句可以跳出for和while的循环体。如果你从for或while循环中终止，任何对应的循环else块将不执行。

continue语句被用来告诉Python跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

循环语句可以有else子句;它在穷尽列表(以for循环)或条件变为假(以while循环)循环终止时被执行,但循环被break终止时不执行.如下查寻质数的循环例子:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # 循环中没有找到元素
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

pass语句

pass语句什么都不做。它只在语法上需要一条语句但程序不需要任何操作时使用.例如:

```
>>> while True:
...     pass  # 等待键盘中断 (Ctrl+C)
```

最小的类:

```
>>> class MyEmptyClass:
...     pass
```

Python 函数

本章节我们将为大家介绍Python中函数的应用。

该章节可参阅Python 函数应用详解。

Python 定义函数使用 def 关键字，一般格式如下：

```
def  函数名（参数列表）：
    函数体
```

让我们使用函数来输出"Hello World！"：

```
>>> def hello() :
    print("Hello World!")

>>> hello()
Hello World!
>>>
```

更复杂点的应用，函数中带上参数变量:

```
def area(width, height):
    return width * height

def print_welcome(name):
    print("Welcome", name)

print_welcome("Fred")
w = 4
h = 5
print("width =", w, " height =", h, " area =", area(w, h))
```

以上实例输出结果：

```
Welcome Fred
width = 4  height = 5  area = 20
```

函数变量作用域

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

通过以下实例，你可以清楚了解Python函数变量的作用域：

```
#!/usr/bin/env python3
a = 4  # 全局变量

def print_func1():
    a = 17 # 局部变量
    print("in print_func a = ", a)
def print_func2():
    print("in print_func a = ", a)
print_func1()
print_func2()
print("a = ", a)
```

以上实例运行结果如下：

```
in print_func a =  17
in print_func a =  4
a =  4
```

关键字参数

函数也可以使用 kwarg=value 的关键字参数形式被调用.例如,以下函数:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

可以以下几种方式被调用:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

以下为错误调用方法：

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

返回值

Python的函数的返回值使用return语句，可以将函数作为一个值赋值给指定变量：

```
def return_sum(x,y):
    c = x + y
    return c

res = return_sum(4,5)
print(res)
```

你也可以让函数返回空值：

```
def empty_return(x,y):
    c = x + y
    return

res = empty_return(4,5)
print(res)
```

可变参数列表

最后,一个最不常用的选择是可以让函数调用可变个数的参数.这些参数被包装进一个元组(查看元组和序列).在这些可变个数的参数之前,可以有零到多个普通的参数:

```
def arithmetic_mean(*args):
    sum = 0
    for x in args:
        sum += x
    return sum

print(arithmetic_mean(45,32,89,78))
print(arithmetic_mean(8989.8,78787.78,3453,78778.73))
print(arithmetic_mean(45,32))
print(arithmetic_mean(45))
print(arithmetic_mean())
```

以上实例输出结果为：

```
244
170009.31
77
45
0
```

更详细教程请参阅参阅Python 函数应用详解。

Python 数据结构

本章节我们主要结合前面所学的知识点来介绍Python数据结构。

列表

Python中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：

方法	描述
list.append(x)	把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]。
list.extend(L)	通过添加指定列表的所有元素来扩充列表，相当于 a[len(a):] = L。
list.insert(i, x)	在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x) 。
list.remove(x)	删除列表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。
list.pop([i])	从列表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop() 返回最后一个元素。元素随即从列表中被删除。（方法中 i 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。）
list.clear()	移除列表中的所有项，等于del a[:]。
list.index(x)	返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。
list.count(x)	返回 x 在列表中出现的次数。
list.sort()	对列表中的元素进行排序。
list.reverse()	倒排列表中的元素。
list.copy()	返回列表的浅复制，等于a[:]。

下面示例演示了列表的大部分方法：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

注意：类似 insert, remove 或 sort 等修改列表的方法没有返回值。

将列表当做堆栈使用

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

将列表当作队列使用

也可以把列表当做队列用，只是在队列里第一加入的元素，第一个取出来；但是拿列表用作这样的目的效率不高。在列表的最后添加或者弹出元素速度快，然而在列表里插入或者从头部弹出速度却不快（因为所有其他的元素都得一个一个地移动）。

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

列表推导式

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

现在我们玩一点小花样：

```
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

这里我们对序列里每一个元素逐个调用某方法：

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

我们可以用 if 子句作为过滤器：

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

以下是一些关于循环和其它技巧的演示：

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

列表推导式可以使用复杂表达式或嵌套函数：

```
>>> [str(round(355/113, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

嵌套列表解析

Python的列表还可以嵌套。

以下实例展示了3X4的矩阵列表：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

以下实例将3X4的矩阵列表转换为4X3列表：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

以下实例也可以使用以下方法来实现：

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

另外一种实现方法：

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

del 语句

使用 del 语句可以从一个列表中依索引而不是值来删除一个元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切割，或清空整个列表（我们以前介绍的方法是给该切割赋一个空列表）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

也可以用 del 删除实体变量：

```
>>> del a
```

元组和序列

元组由若干逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号， 不过括号通常是必须的（如果元组是更大的表达式的一部分）。

集合

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {} ；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

集合也支持推导式：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

字典

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键=>值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{}。

这是一个字典运用的简单例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

构造函数 dict() 直接从键值对元组列表中构建字典。如果有固定的模式，列表推导式指定特定的键值对：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典推导可以用来创建任意键和值的表达式词典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果关键字只是简单的字符串，使用关键字参数指定键值对有时候更方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

遍历技巧

在字典中遍历时，关键字和对应的值可以使用 items() 方法同时解读出来：

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中遍历时，索引位置和对应值可以使用 enumerate() 函数同时得到：

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

同时遍历两个或更多的序列，可以使用 zip() 组合：

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要反向遍历一个序列，首先指定这个序列，然后调用 reversesd() 函数：

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

要按顺序遍历一个序列，使用 sorted() 函数返回一个已排序的序列，并不修改原值：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

其他参阅文档(Python2.x)

- Python 列表
- Python 元组
- Python 字典

Python 模块

在前面的几个章节中我们脚本上是用python解释器来编程，如果你从Python解释器退出再进入，那么你定义的所有的方法和变量就都消失了。

为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块。

模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用python标准库的方法。下面是一个使用python标准库中模块的例子。

```
#!/usr/bin/python3
# Filename: using_sys.py

import sys

print('命令行参数如下:')
for i in sys.argv:
    print(i)

print('/n/nThe PYTHONPATH is', sys.path, '/n')
```

执行结果如下所示：

```
E:\python33\src>python using_sys.py 参数1 参数2
命令行参数如下:
using_sys.py
参数1
参数2
/n/nThe PYTHONPATH is ['E:\\python33\\src', 'C:\\Windows\\system32\\python33.zip
', 'E:\\python33\\DLLs', 'E:\\python33\\lib', 'E:\\python33', 'E:\\python33\\lib
\\site-packages'] /n
```

- 1、import sys引入python标准库中的sys.py模块；这是引入某一模块的方法。
- 2、sys.argv是一个包含命令行参数的列表。
- 3、sys.path包含了一个Python解释器自动查找所需模块的路径的列表。

当我们使用import语句的时候，Python解释器是怎样找到对应的文件的呢？

这就涉及到Python的搜索路径，搜索路径是由一系列目录名组成的，Python解释器就依次从这些目录中去寻找锁引入的模块。

这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。

搜索路径是在Python编译或安装的时候确定的，安装新的库应该也会修改。搜索路径被存储在sys模块中的path变量，做一个简单的实验，在交互式解释器中，输入以下代码：

```
import sys
sys.path
```

输出结果：

```
>>> sys.path
['', 'E:\\python33\\Lib\\idlelib', 'C:\\Windows\\system32\\python33.zip', 'E:\\python33\\
```

sys.path输出是一个列表，其中第一项是空串''，代表当前目录（若是从一个脚本中打印出来的话，可以更清楚地看出是哪个目录），亦即我们执行python解释器的目录（对于脚本的话就是运行的脚本所在的目录）。

因此若像我一样在当前目录下存在与要引入模块同名的文件，就会把要引入的模块屏蔽掉。

了解了搜索路径的概念，就可以在脚本中修改sys.path来引入一些不在搜索路径中的模块。

现在，在解释器的当前目录或者sys.path中的一个目录里面来创建一个fibo.py的文件，代码如下：

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

然后进入Python解释器，使用下面的命令导入这个模块：

```
>>> import fibo
```

这样做并没有把直接定义在fibo中的函数名称写入到当前符号表里，只是把模块fibo的名字写到了那里。

可以使用模块名称来访问函数：

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果你打算经常使用一个函数，你可以把它赋给一个本地的名称：

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

深入模块

模块除了方法定义，还可以包括可执行的代码。这些代码一般用来初始化这个模块。这些代码只有在第一次被导入时才会被执行。

每个模块有各自独立的符号表，在模块内部为所有的函数当作全局符号表来使用。

所以，模块的作者可以放心大胆的在模块内部使用这些全局变量，而不用担心把其他用户的全局变量搞花。

从另一个方面，当你确实知道你在做什么的话，你也可以通过 modname.itemname 这样的表示法来访问模块内的函数。

模块是可以导入其他模块的。在一个模块（或者脚本，或者其他地方）的最前面使用 import 来导入一个模块，当然这只是一个惯例，而不是强制的。被导入的模块的名称将被放入当前操作的模块的符号表中。

还有一种导入的方法，可以使用 import 直接把模块内（函数，变量的）名称导入到当前操作模块。比如:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种导入的方法不会把被导入的模块的名称放在当前的字符表中（所以在这个例子里面，fibo 这个名称是没有定义的）。

这还有一种方法，可以一次性的把模块中的所有（函数，变量）名称都导入到当前模块的字符表:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这将把所有的名字都导入进来，但是那些由单一下划线（_）开头的名字不在此例。大多数情况， Python程序员不使用这种方法，因为引入的其它来源的命名，很可能覆盖了已有的定义。

name属性

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用**name**属性来使该程序块仅在该模块自身运行时执行。

```
#!/usr/bin/python3
# Filename: using_name.py

if __name__ == '__main__':
    print('程序自身在运行')
else:
    print('我来自另一模块')
```

运行输出如下：

```
$ python using_name.py
```

程序自身在运行

```
$ python
>>> import using_name
我来自另一模块
>>>
```

说明： 每个模块都有一个**name**属性，当其值是**'main'**时，表明该模块自身在运行，否则是被引入。

dir() 函数

```
内置的函数 dir() 可以找到模块内定义的所有名称。以一个字符串列表的形式返回:
</p>
<pre>
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

如果没有给定参数，那么 dir() 函数会罗列出当前定义的所有名称:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir() # 得到一个当前模块中定义的属性列表
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
>>> a = 5 # 建立一个新的变量 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # 删除变量名a
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
>>>
```

标准模块

Python 本身带着一些标准的模块库，在 Python 库参考文档中将会介绍到（就是后面的"库参考文档"）。

有些模块直接被构建在解析器里，这些虽然不是一些语言内置的功能，但是他却能很高效的使用，甚至是系统级调用也没问题。

这些组件会根据不同的操作系统进行不同形式的配置，比如 winreg 这个模块就只会提供给 Windows 系统。

应该注意到这有一个特别的模块 sys ，它内置在每一个 Python 解析器中。变量 sys.ps1 和 sys.ps2 定义了主提示符和副提示符所对应的字符串:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

包

包是一种管理 Python 模块命名空间的形式，采用"点模块名称"。

比如一个模块的名称是 A.B， 那么他表示一个包 A中的子模块 B 。

就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，采用点模块名称这种形式也不用担心不同库之间的模块重名的情况。

这样不同的作者都可以提供 NumPy 模块，或者是 Python 图形库。

不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个"包"）。

现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如：.wav，:file:.aiff，:file:.au，），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。

并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

在导入一个包的时候，Python 会根据 sys.path 中的目录来寻找这个包中包含的子目录。

目录只有包含一个叫做 **init**.py 的文件才会被认作是一个包，主要是为了避免一些滥俗的名字（比如叫做 string）不小心的影响搜索路径中的有效模块。

最简单的情况，放一个空的 :file:**init**.py就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的） **all**变量赋值。

用户可以每次只导入一个包里面的特定模块，比如:

```
import sound.effects.echo
```

这将会导入子模块:mod:song.effects.echo。 他必须使用全名去访问:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种导入子模块的方法是:

```
from sound.effects import echo
```

这同样会导入子模块:mod:echo，并且他不需要那些冗长的前缀，所以他可以这样使用:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种变化就是直接导入一个函数或者变量:

```
from sound.effects.echo import echofilter
```

同样的，这种方法会导入子模块:mod:echo，并且可以直接使用他的:func:echofilter函数:

echofilter(input, output, delay=0.7, atten=4)

注意当使用from package import item这种形式的时候，对应的item既可以是包里面的子模块（子包），或者包里面定义的其他名称，比如函数，类或者变量。

import语法会首先把item当作一个包定义的名称，如果没找到，再试图按照一个模块去导入。如果还没找到，恭喜，一个:exc:ImportError 异常被抛出了。

反之，如果使用形如import item.subitem.subsubitem这种导入形式，除了最后一项，都必须是包，而最后一项则可以是模块或者是包，但是不可以是类，函数或者变量的名字。

从一个包中导入*

设想一下，如果我们使用 from sound.effects import *会发生什么？

Python 会进入文件系统，找到这个包里面所有的子模块，一个一个的把它们都导入进来。

但是很不幸，这个方法在 Windows平台上工作的就不是非常好，因为Windows是一个大小写不区分的系统。

在这类平台上，没有人敢担保一个叫做 ECHO.py 的文件导入为模块:mod:echo还是:mod:Echo甚至:mod:ECHO。

（例如，Windows 95就很讨厌的把每一个文件的首字母大写显示）而且 DOS 的 8+3 命名规则对长模块名称的处理会把问题搞得更纠结。

为了解决这个问题，只能烦劳包作者提供一个精确的包的索引了。

导入语句遵循如下规则：如果包定义文件 **init**.py 存在一个叫做 **all** 的列表变量，那么在使用 from package import * 的时候就把这个列表中的所有名字作为包内容导入。

作为包的作者，可别忘了在更新包之后保证 **all** 也更新了啊。你说我就不这么做，我就不使用导入*这种用法，好吧，没问题，谁让你是老板呢。这里有一个例子，在:file:sounds/effects/**init**.py中包含如下代码:

```
__all__ = ["echo", "surround", "reverse"]
```

这表示当你使用from sound.effects import *这种用法时，你只会导入包里面这三个子模块。

如果**all**真的而没有定义，那么使用from sound.effects import 这种语法的时候，就不会*导入包:mod:sound.effects里的任何子模块。他只是把包:mod:sound.effects和它里面定义的所有内容导入进来（可能运行:file:**init**.py里定义的初始化代码）。

这会把 :file:**init**.py里面定义的所有名字导入进来。并且他不会破坏掉我们在这句话之前导入的所有明确指定的模块。看下这部分代码:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

这个例子中，在执行from...import前，包:mod:sound.effects中的echo和surround模块都被导入到当前的命名空间中了。（当然如果定义了**all**就更没问题了）

通常我们并不主张使用*这种方法来导入模块，因为这种方法经常会导致代码的可读性降低。不过这样倒的确是可以省去不少敲键的功夫，而且一些模块都设计成了只能通过特定的方法导入。

记住，使用from Package import specific_submodule这种方法永远不会有错。事实上，这也是推荐的方法。除非是你要导入的子模块有可能和其他包的子模块重名。

如果在结构中包是一个子包（比如这个例子中对于包:mod:sound来说），而你又想导入兄弟包（同级别的包）你就得使用导入绝对的路径来导入。比如，如果模块:mod:sound.filters.vocoder 要使用包:mod:sound.effects中的模块:mod:echo，你就要写成 from sound.effects import echo。

```
from . import echo
from .. import formats
from ..filters import equalizer
```

无论是隐式的还是显式的相对导入都是从当前模块开始的。主模块的名字永远是"**main**"，一个Python应用程序的主模块，应当总是使用绝对路径引用。

包还提供一个额外的属性，:attr:**path**。这是一个目录列表，里面每一个包含的目录都有为这个包服务的:file:**init**.py，你得在其他:file:**init**.py被执行前定义哦。可以修改这个变量，用来影响包含在包里面的模块和子包。

这个功能并不常用，一般用来扩展包里面的模块。

Python 输入和输出

在前面几个章节中，我们其实已经接触了 Python 的输入输出的功能。本章节我们将具体介绍 Python 的输入输出。

输出格式美化

Python两种输出值的方式: 表达式语句 和 print() 函数。(第三种方式是使用文件对象的 write() 方法; 标准输出文件可以用 sys.stdout 引用。)

如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。

str() 函数返回一个用户易读的表达形式。

repr() 产生一个解释器易读的表达形式。

例如

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

这里有两种方式输出一个平方与立方的表:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line 注意前一行 'end' 的使用
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

注意：在第一个例子中, 每列间的空格由 print() 添加。

这个例子展示了字符串对象的 rjust() 方法, 它可以将字符串靠右, 并在左边填充空格。

还有类似的方法, 如 ljust() 和 center()。 这些方法并不会写任何东西, 它们仅仅返回新的字符串。

另一个方法 zfill(), 它会在数字的左边填充 0，如下所示：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format() 的基本使用如下:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。

在括号中的数字用于指向传入对象在 format() 中的位置，如下所示：

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

如果在 format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

位置及关键字参数可以任意的结合:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                       other='Georg'))
The story of Bill, Manfred, and Georg.
```

'!a' (使用 ascii()), '!s' (使用 str()) 和 '!r' (使用 repr()) 可以用于在格式化某个值之前对其进行转化:

```
>>> import math
>>> print('The value of PI is approximately {}.'.format(math.pi))
The value of PI is approximately 3.14159265359.
>>> print('The value of PI is approximately {!r}.'.format(math.pi))
The value of PI is approximately 3.141592653589793.
```

可选项 ':' 和格式标识符可以跟着字段名。 这就允许对值进行更好的格式化。 下面的例子将 Pi 保留到小数点后三位：

```
>>> import math
>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))
The value of PI is approximately 3.142.
```

在 ':' 后传入一个整数, 可以保证该域至少有这么多的宽度。 用于美化表格时很有用。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

如果你有一个很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情。

最简单的就是传入一个字典, 然后使用方括号 '[]' 来访问键值：

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
          'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

也可以通过在 table 变量前使用 '**' 来实现相同的功能：

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

旧式字符串格式化

% 操作符也可以实现字符串格式化。 它将左边的参数作为类似 sprintf() 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

```
>>> import math
>>> print('The value of PI is approximately %5.3f.' % math.pi)
The value of PI is approximately 3.142.
```

因为 str.format() 比较新的函数， 大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 str.format().

读和写文件

open() 将会返回一个 file 对象，基本语法格式如下:

```
open(filename, mode)
```

实例:

```
>>> f = open('/tmp/workfile', 'w')
```

- 第一个参数为要打开的文件名。
- 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

文件对象的方法

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

f.read()

为了读取一个文件的内容， 调用 f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回。

size 是一个可选的数字类型的参数。 当 size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline()

f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串, 说明已经已经读取到最后一行。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

f.readlines()

f.readlines() 将返回该文件中包含的所有行。

如果设置可选参数 sizehint, 则读取指定长度的字节, 并且将这些字节按行分割。

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

另一种方式是迭代一个文件对象然后读取每行:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

这个方法很简单, 但是并没有提供一个很好的控制。 因为两者的处理机制不同, 最好不要混用。

f.write()

f.write(string) 将 string 写入到文件中, 然后返回写入的字符数。

```
>>> f.write('This is a test\n')
15
```

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

f.tell()

f.tell() 返回文件对象当前所处的位置, 它是从文件开头开始算起的字节数。

f.seek()

如果要改变文件当前的位置, 可以使用 f.seek(offset, from_what)函数， from_what 表示开始读取的位置，offset表示从from_what再移动一定量的距离，比如f.seek(10, 3)表示定位到第三个字符并再后移10个字符。

from_what值为0时表示文件的开始，它也可以省略，缺省是0即文件开头。下面给出一个完整的例子：

```
>>> f = open('/tmp/workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # 移动到文件的第六个字节
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # 移动到文件的倒数第三字节
13
>>> f.read(1)
b'd'
```

f.close()

在文本文件中 (那些打开文件的模式下没有 b 的), 只会相对于文件起始位置进行定位。

当你处理完一个文件后, 调用 f.close() 来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
<pre>
<p>
当处理一个文件对象时, 使用 with 关键字是非常好的方式。在结束后, 它会帮你正确的关闭文件。 而且写起来也比
<pre>
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有其他方法, 如 isatty() 和 trucate(), 但这些通常比较少用。

pickle 模块

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过pickle模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

注解：从 file 中读取一个字符串，并将它重构为原来的python对象。

file: 类文件对象，有read()和readline()接口。

实例1：

```
#使用pickle模块将数据对象保存到文件

import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

实例2：

```
#使用pickle模块从文件中重构python对象

import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Python 错误和异常

作为Python初学者，在刚学习Python编程时，经常会看到一些报错信息，在前面我们没有提及，这章节我们会专门介绍。

Python有两种错误很容易辨认：语法错误和异常。

语法错误

Python 的语法错误或者称之为解析错，是初学者经常碰到的，如下实例

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号（:）。

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

异常

即便Python程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

异常以不同的类型出现，这些类型都作为信息的一部分打印出来: 例子中的类型有 ZeroDivisionError，NameError 和 TypeError。

错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

异常处理

以下例子中，让用户输入一个合法的整数，但是允许用户中断这个程序（使用 Control-C 或者操作系统提供的方法）。用户中断的信息会引发一个 KeyboardInterrupt 异常。

```
>>> while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except ValueError:
            print("Oops!  That was no valid number.  Try again   ")
```

try语句按照如下方式工作；

- 首先，执行try子句（在关键字try和关键字except之间的语句）
- 如果没有异常发生，忽略except子句，try子句执行后结束。
- 如果在执行try子句的过程中发生了异常，那么try子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的except子句将被执行。最后执行 try 语句之后的代码。
- 如果一个异常没有与任何的except匹配，那么这个异常将会传递给上层的try中。

一个 try 语句可能包含多个except子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。

处理程序将只针对对应的try子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。

一个except子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组，例如:

```
    except (RuntimeError, TypeError, NameError):
        pass
```

最后一个except子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出。

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try except 语句还有一个可选的else子句，如果使用这个子句，那么必须放在所有的except子句之后。这个子句将在try子句没有发生任何异常的时候执行。例如:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 else 子句比把所有的语句都放在 try 子句里面要好，这样可以避免一些意想不到的、而except又没有捕获的异常。

异常处理并不仅仅处理那些直接发生在try子句中的异常，而且还能处理子句中调用的函数（甚至间接调用的函数）里抛出的异常。例如:

```
>>> def this_fails():
        x = 1/0

>>> try:
        this_fails()
    except ZeroDivisionError as err:
        print('Handling run-time error:', err)

Handling run-time error: int division or modulo by zero
```

抛出异常

Python 使用 raise 语句抛出一个指定的异常。例如:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

```
>>> try:
        raise NameError('HiThere')
    except NameError:
        print('An exception flew by!')
        raise

An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

用户自定义异常

你可以通过创建一个新的exception类来拥有自己的异常。异常应该继承自 Exception 类，或者直接继承，或者间接继承，例如:

```
>>> class MyError(Exception):
        def __init__(self, value):
            self.value = value
        def __str__(self):
            return repr(self.value)

>>> try:
        raise MyError(2*2)
    except MyError as e:
        print('My exception occurred, value:', e.value)

My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在这个例子中，类 Exception 默认的 **init**() 被覆盖。

当创建一个模块有可能抛出多种不同的异常时，一种通常的做法是为这个包建立一个基础异常类，然后基于这个基础类为不同的错误情况创建不同的子类:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

大多数的异常的名字都以"Error"结尾，就跟标准的异常命名一样。

定义清理行为

try 语句还有另外一个可选的子句，它定义了无论在任何情况下都会执行的清理行为。 例如:

```
>>> try:
        raise KeyboardInterrupt
    finally:
        print('Goodbye, world!')

Goodbye, world!
KeyboardInterrupt
```

以上例子洪不管try子句里面有没有发生异常，finally子句都会执行。

如果一个异常在 try 子句里（或者在 except 和 else 子句里）被抛出，而又没有任何的 except 把它截住，那么这个异常会在 finally 子句执行后再次被抛出。

下面是一个更加复杂的例子（在同一个 try 语句里包含 except 和 finally 子句）:

```
>>> def divide(x, y):
        try:
            result = x / y
        except ZeroDivisionError:
            print("division by zero!")
        else:
            print("result is", result)
        finally:
            print("executing finally clause")

>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

预定义的清理行为

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。

这面这个例子展示了尝试打开一个文件，然后把内容打印到屏幕上:

```
for line in open("myfile.txt"):
    print(line, end="")
```

以上这段代码的问题是，当执行完毕后，文件会保持打开状态，并没有被关闭。

关键词 with 语句就可以保证诸如文件之类的对象在使用完之后一定会正确的执行他的清理方法:

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

以上这段代码执行完毕后，就算在处理过程中出问题了，文件 f 总是会关闭。

Python 类

和其它编程语言相比，Python 在尽可能不增加新的语法和语义的情况下加入了类机制。

Python中的类提供了面向对象编程的所有基本功能：类的继承机制允许多个基类，派生类可以覆盖基类中的任何方法，方法中可以调用基类中的同名方法。

对象可以包含任意数量和类型的数据。

类定义

语法格式如下：

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

类实例化后，可以使用其属性，实际上，创建一个类之后，可以通过类名访问其属性。

类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用和 Python 中所有的属性引用一样的标准语法：obj.name。

类对象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样:

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

实例化类：

```
x = MyClass()
```

以上创建了一个新的类实例并将该对象赋给局部变量 x，x 为空的对象。

很多类都倾向于将对象创建为有初始状态的。因此类可能会定义一个名为 **init()** 的特殊方法（构造方法），像下面这样：

```
def __init__(self):
    self.data = []
```

类定义了 **init()** 方法的话，类的实例化操作会自动调用 **init()** 方法。所以在下例中，可以这样创建一个新的实例:

```
x = MyClass()
```

当然， **init()** 方法可以有参数，参数通过 **init()** 传递到类的实例化操作上。例如:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

类的方法

在类地内部，使用def关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数self,且为第一个参数:

```
#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s is speaking: I am %d years old" %(self.name,self.age))

p = people('tom',10,30)
p.speak()
```

继承

Python 同样支持类的继承，如果一种语言不支持继承就，类就没有什么意义。派生类的定义如下所示:

```
class DerivedClassName(BaseClassName1):
    <statement-1>
    .
    .
    .
    <statement-N>
```

需要注意圆括号中基类的顺序，若是基类中有相同的方法名，而在子类使用时未指定，python从左至右搜索 即方法在子类中未找到时，从左到右查找基类中是否包含方法。

BaseClassName（示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用:

```
class DerivedClassName(modname.BaseClassName):
```

实例

```
#单继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构函
        people.__init__(self,n,a,w)
        self.grade = g
    #覆写父类的方法
    def speak(self):
        print("%s is speaking: I am %d years old,and I am in grade %d"%(self.name,self.ag

s = student('ken',20,60,3)
s.speak()
```

多重继承

Python同样有限的支持多继承形式。多继承的类定义形如下例:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

需要注意圆括号中父类的顺序，若是父类中有相同的方法名，而在子类使用时未指定，python从左至右搜索 即方法在子类中未找到时，从左到右查找父类中是否包含方法。

```
#另一个类，多重继承之前的准备
class speaker():
    topic = ''
    name = ''
    def __init__(self,n,t):
        self.name = n
        self.topic = t
    def speak(self):
        print("I am %s,I am a speaker!My topic is %s"%(self.name,self.topic))

#多重继承
class sample(speaker,student):
    a =''
    def __init__(self,n,a,w,g,t):
        student.__init__(self,n,a,w,g)
        speaker.__init__(self,n,t)

test = sample("Tim",25,80,4,"Python")
test.speak()#方法名同，默认调用的是在括号中排前地父类的方法
```

类私有方法

__private_method 两个下划线开头，声明该方法为私有方法，不能在类地外部调用。

在类的内部调用slef.__private_methods。

类的专有方法：

- **init** 构造函数，在生成对象时调用
- **del** 析构函数，释放对象时使用
- **repr** 打印，转换
- **setitem**按照索引赋值
- **getitem**按照索引获取值
- **len**获得长度
- **cmp**比较运算
- **call**函数调用
- **add**加运算
- **sub**减运算
- **mul**乘运算
- **div**除运算
- **mod**求余运算
- **pow**称方

更多介绍请查看：http://www.w3cschool.cc/python/python-object.html

Python 标准库概览

操作系统接口

os模块提供了不少与操作系统相关联的函数。

```
>>> import os
>>> os.getcwd()      # 返回当前的工作目录
'C:\\Python34'
>>> os.chdir('/server/accesslogs')   # 修改当前的工作目录
>>> os.system('mkdir today')   # 执行系统命令 mkdir
0
```

建议使用 "import os" 风格而非 "from os import *"。这样可以保证随操作系统不同而有所变化的 os.open() 不会覆盖内置函数 open()。

在使用 os 这样的大型模块时内置的 dir() 和 help() 函数非常有用:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

针对日常的文件和目录管理任务，:mod:shutil 模块提供了一个易于使用的高级接口:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

文件通配符

glob模块提供了一个函数用于从目录通配符搜索中生成文件列表:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

命令行参数

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于 sys 模块的 argv 变量。例如在命令行中执行 "python demo.py one two three" 后可以得到以下输出结果:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

错误输出重定向和程序终止

sys 还有 stdin，stdout 和 stderr 属性，即使在 stdout 被重定向时，后者也可以用于显示警告和错误信息。

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

大多脚本的定向终止都使用 "sys.exit()"。

字符串正则匹配

re模块为高级字符串处理提供了正则表达式工具。对于复杂的匹配和处理，正则表达式提供了简洁、优化的解决方案:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

如果只需要简单的功能，应该首先考虑字符串方法，因为它们非常简单，易于阅读和调试:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

数学

math模块为浮点运算提供了对底层C函数库的访问:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random提供了生成随机数的工具。

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

访问 互联网

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从 urls 接收的数据的 urllib.request 以及用于发送电子邮件的 smtplib:

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     line = line.decode('utf-8')  # Decoding the binary data to text.
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

注意第二个例子需要本地有一个在运行的邮件服务器。

日期和时间

datetime模块为日期和时间处理同时提供了简单和复杂的方法。

支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。

该模块还支持时区处理:

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03\. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

数据压缩

以下模块直接支持通用的数据打包和压缩格式：zlib，gzip，bz2，zipfile，以及 tarfile。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多,timeit 证明了现代的方法更快一些。

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

相对于 timeit 的细粒度，:mod:profile 和 pstats 模块提供了针对更大代码块的时间度量工具。

测试模块

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试

doctest模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。

测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。

通过用户提供的例子，它强化了文档，允许 doctest 模块确认代码的结果是否与文档一致:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # 自动验证嵌入测试
```

unittest模块不像 doctest模块那么容易使用，不过它可以在一个独立的文件里提供一个更全面的测试集:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

Django 教程

Django 安装

在安装 Django 前，系统需要已经安装了Python的开发环境。接下来我们来具体看下不同系统下Django的安装。

Window 下安装 Django

如果你还未安装Python环境需要先下载Python安装包。

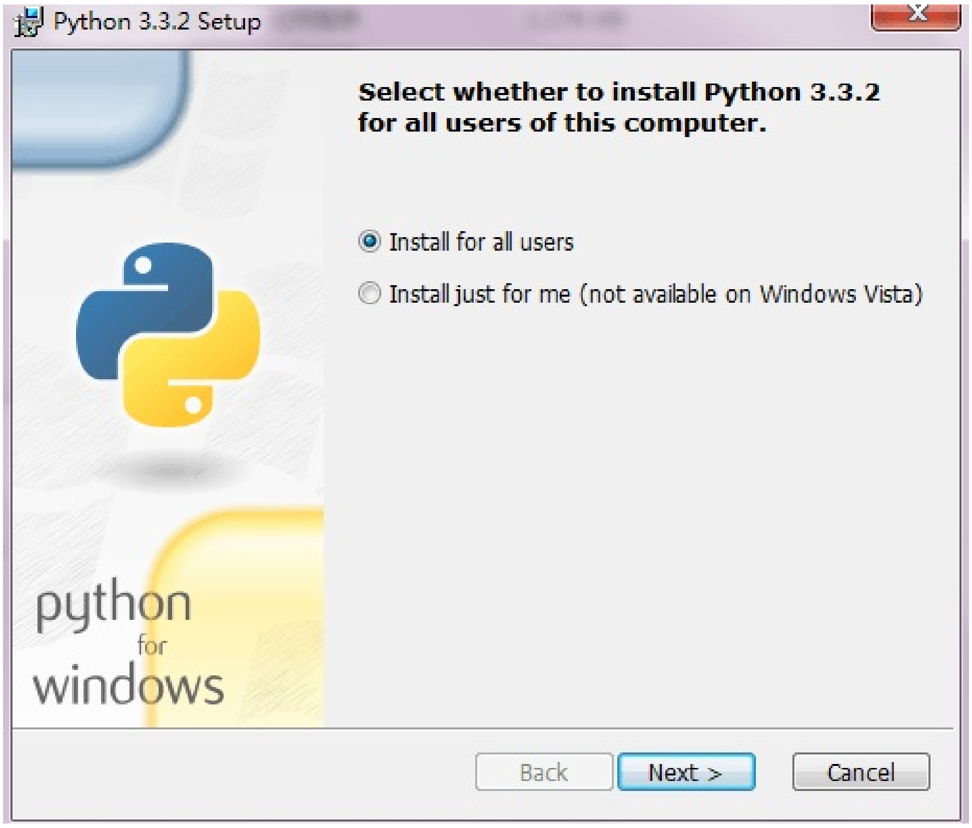
1、Python 下载地址：https://www.python.org/downloads/

2、Django 下载地址：https://www.djangoproject.com/download/

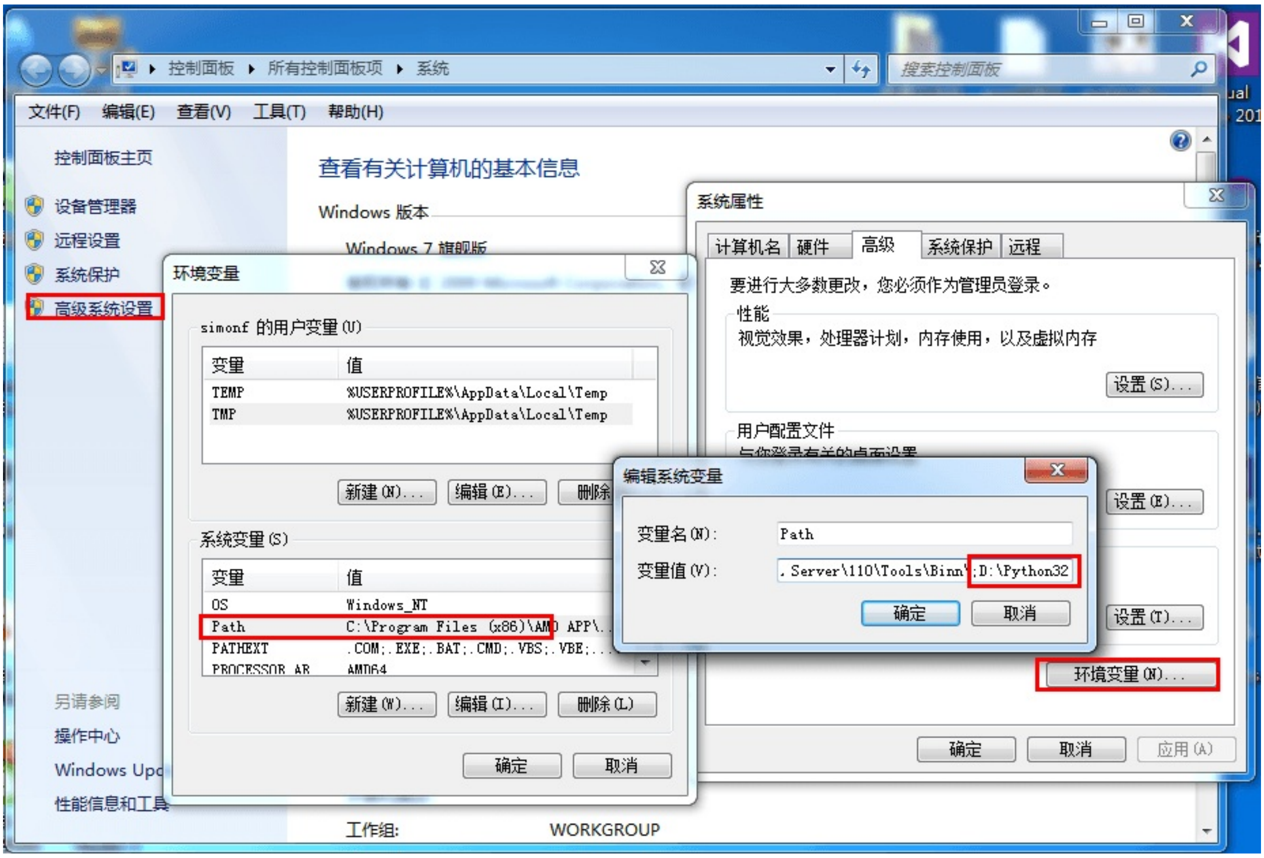
注意：目前Django 1.6.x以上版本已经完全兼容Python 3.x。

Python 安装(已安装的可跳过)

安装Python你只需要下载python-x.x.x.msi文件，然后一直点击"Next"按钮即可。

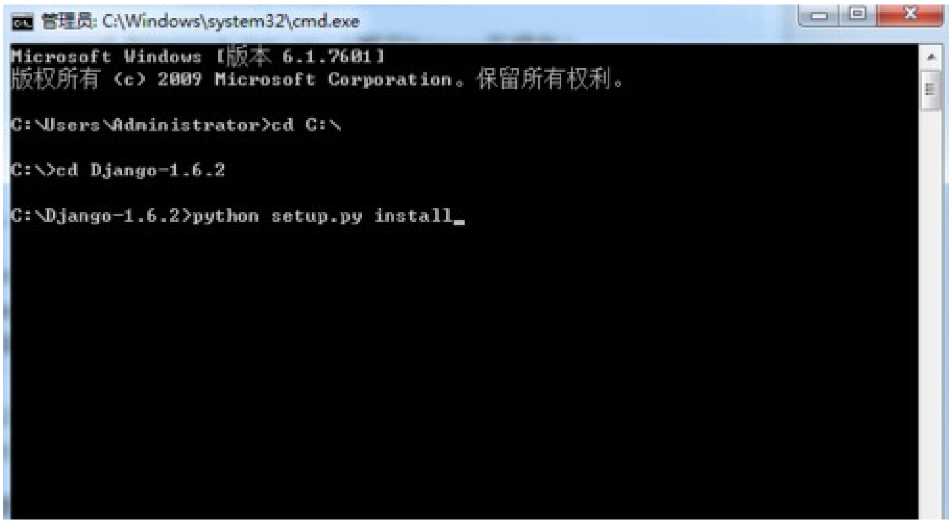


安装完成后你需要设置Python环境变量。 右击计算机->属性->高级->环境变量->修改系统变量path，添加Python安装地址，本文实例使用的是C:\Python33，你需要根据你实际情况来安装。

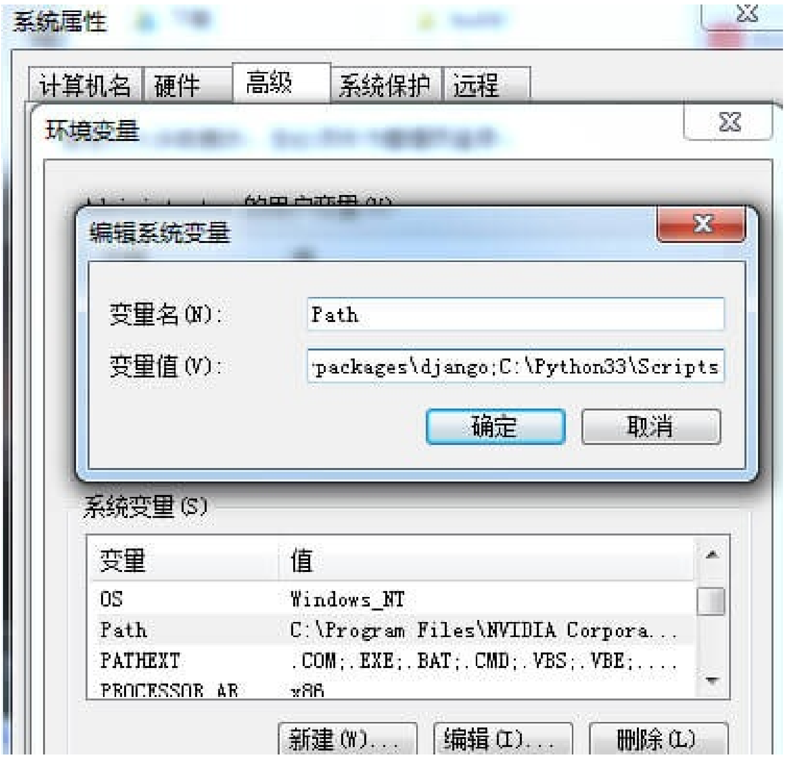


Django 安装

下载 Django 压缩包，解压并和Python安装目录放在同一个根目录，进入 Django 目录，执行python setup.py install，然后开始安装，Django将要被安装到Python的Lib下site-packages。



然后是配置环境变量，将这几个目录添加到系统环境变量中： C:/Python33/Lib/site-packages/django;C:/Python33/Scripts。 添加完成后就可以使用Django的django-admin.py命令新建工程了。



检查是否安装成功

输入以下命令进行检查:

```
python import django
django.get_version()
```

如果输出了Django的版本号说明安装正确。

Linux 上安装 Django

yum 安装方法

以下安装位于 Centos Linux 环境下安装，如果是你的 Linux 系统是 ubuntu 请使用 apt-get 命令。

默认情况下 Linux 环境已经支持了Python。你可以在终端输入Python命令来查看是否已经安装。

```
Python  2.7.3  (default,  Aug  1  2012,  05:14:39)  [GCC 4.6.3] on linux2 Type  "help",
```

安装 setuptools

命令：

```
yum install setuptools
```

完成之后，就可以使用 easy_install 命令安装 django

```
easy_install django
```

之后我们在python解释器输入以下代码：

```
[root@solar django]# python Python  2.7.3  (default,  May  15  2014,  14:49:08)  [GCC 4.8
```

我们可以看到输出了Django的版本号，说明安装成功。

pip 命令安装方法

```
pip install Django
```

源码安装方法

下载源码包：https://www.djangoproject.com/download/

输入以下命令并安装：

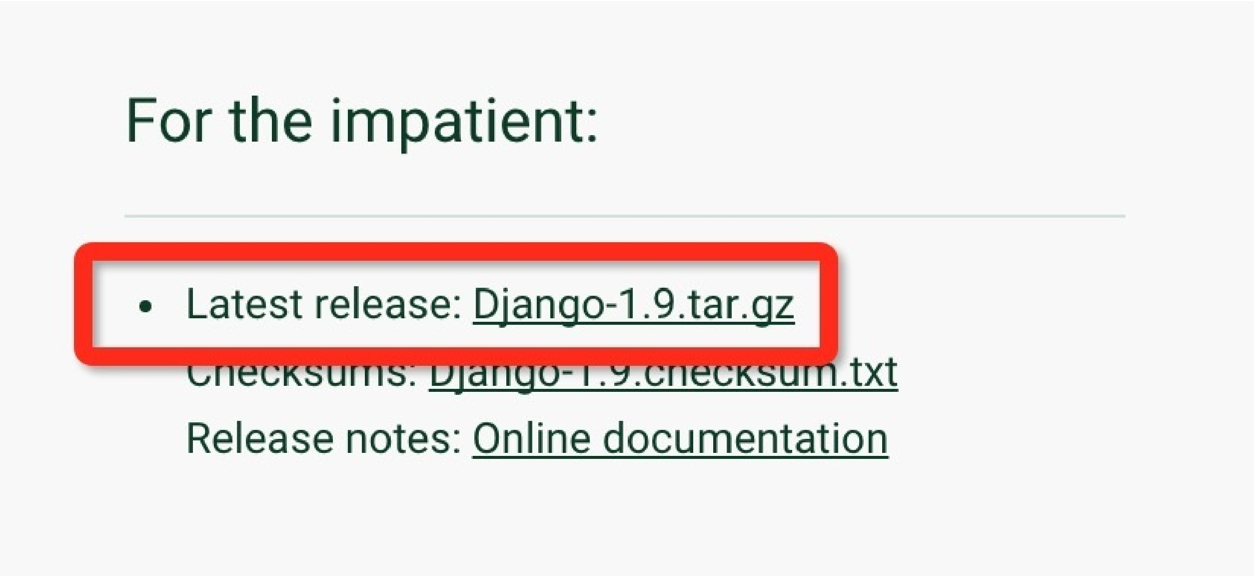
```
tar xzvf Django-X.Y.tar.gz # 解压下载包 cd Django-X.Y # 进入 Django 目录 python setup.py ins
```

安装成功后 Django 位于 Python 安装目录的 site-packages 目录下。

Mac 下安装

下载

从这里下载最新的稳定版本：DJango-1.x.y.tar.gz，在页面右侧列表下载，如下图：



记住是最新的官方版本哦.其中x.y是版本号。进入你下载该文件的文件夹目录，执行如下命令:（Mac下默认是/Users/xxx/Downloads，xxx是你的用户名）

```
$ tar zxvf Django-1.x.y.tar.gz
```

你也可以从 Github 上下载最新版，地址：https://github.com/django/django：

```
git clone https://github.com/django/django.git
```

安装

进入解压后的目录：

```
cd Django-1.x.y
sudo python setup.py install
```

安装成功后会输出以下信息：

```
…… Processing dependencies for  Django==1.x.y Finished processing dependencies for  Djan
```

再进入我们的站点目录，创建 Django 项目：

```
$ django-admin.py startproject testdj
```

启动服务：

```
cd testdj # 切换到我们创建的项目 $ python manage.py runserver …… Starting development serve
```

以上信息说明，项目已启动，访问地址为http://127.0.0.1:8000/。

Django 创建第一个项目

本章我们将介绍Django 管理工具及如何使用 Django 来创建项目，第一个项目我们以 HelloWorld 来命令项目。

Django 管理工具

安装 Django 之后，您现在应该已经有了可用的管理工具 django-admin.py。我们可以使用 django-admin.py 来创建一个项目:

我们可以来看下django-admin.py的命令介绍:

```
[root@solar ~]# django-admin.py
Usage: django-admin.py subcommand [options] [args]

Options:
  -v VERBOSITY, --verbosity=VERBOSITY
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings=SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath=PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Raise on exception
  --version             show program's version number and exit
  -h, --help            show this help message and exit

Type 'django-admin.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[django]
    check
    cleanup
    compilemessages
    createcachetable
......省略部分......
```

创建第一个项目

使用 django-admin.py 来创建 HelloWorld 项目：

```
django-admin.py startproject HelloWorld
```

创建完成后我们可以查看下项目的目录结构：

```
[root@solar ~]# cd HelloWorld/
[root@solar HelloWorld]# tree
.
|-- HelloWorld
|   |-- __init__.py
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
`-- manage.py
```

目录说明：

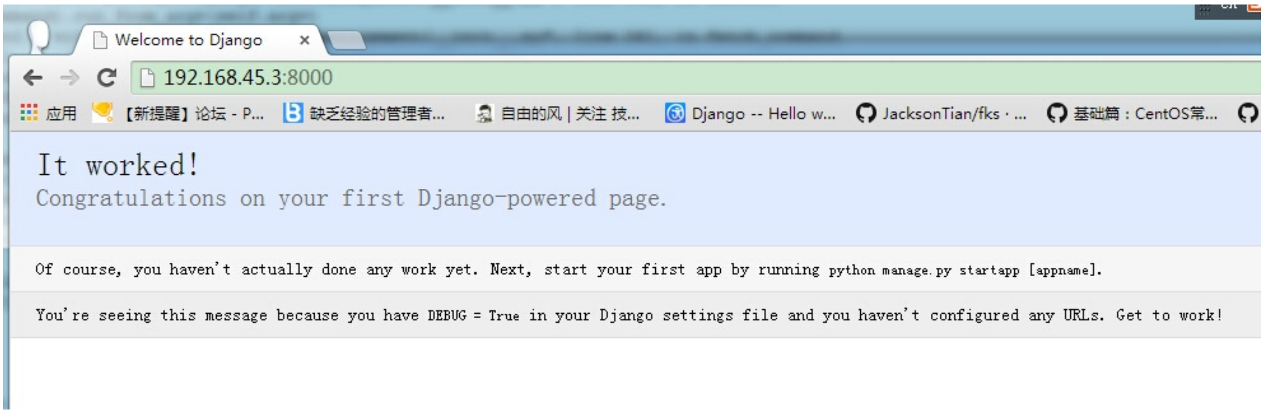
- **HelloWorld:** 项目的容器。
- **manage.py:** 一个实用的命令行工具，可让你以各种方式与该 Django 项目进行交互。
- **HelloWorld/init.py:** 一个空文件，告诉 Python 该目录是一个 Python 包。
- **HelloWorld/settings.py:** 该 Django 项目的设置/配置。
- **HelloWorld/urls.py:** 该 Django 项目的 URL 声明; 一份由 Django 驱动的网站"目录"。
- **HelloWorld/wsgi.py:** 一个 WSGI 兼容的 Web 服务器的入口，以便运行你的项目。

接下来我们进入 HelloWorld 目录输入以下命令，启动服务器：

```
python manage.py runserver 0.0.0.0:8000
```

0.0.0.0让其它电脑可连接到开发服务器，8000为端口号。如果不说明，那么端口号默认为8000。

在浏览器输入你服务器的ip及端口号，如果正常启动，输出结果如下：



视图和 URL 配置

在先前创建的 HelloWorld 目录下的 HelloWorld 目录新建一个 view.py 文件，并输入代码：

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world ! ")
```

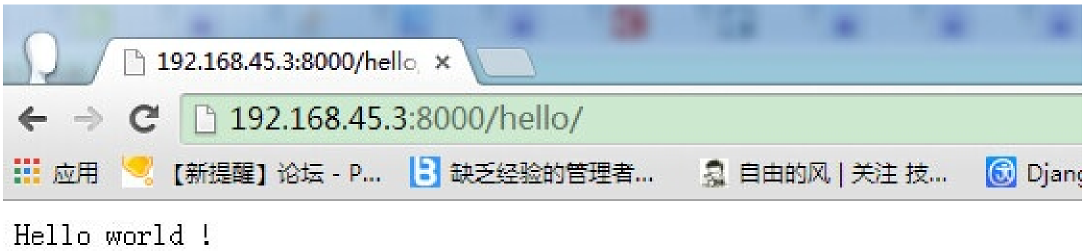
接着，绑定 URL 与视图函数。打开 urls.py 文件，删除原来代码，将以下代码复制粘贴到 urls.py 文件中：

```
from django.conf.urls import *
from HelloWorld.view import hello

urlpatterns = patterns("",
    ('^hello/$', hello),
)
```

整个目录结构如下：

```
[root@solar HelloWorld]# tree
.
|-- HelloWorld
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- settings.py
|   |-- settings.pyc
|   |-- urls.py              # url 配置
|   |-- urls.pyc
|   |-- view.py              # 添加的视图文件
|   |-- view.pyc             # 编译后的视图文件
|   |-- wsgi.py
|   `-- wsgi.pyc
`-- manage.py
```

完成后，启动 Django 开发服务器，并在浏览器访问打开浏览器并访问：



注意：项目中如果代码有改动，服务器会自动监测代码的改动并自动重新载入，所以如果你已经启动了服务器则不需手动重启。

Django 模板

在上一章节中我们使用 django.http.HttpResponse() 来输出"Hello World！"。该方式将数据与视图混合在一起，不符合Django的MVC思想。

本章节我们将为大家详细介绍 Django 模板的应用，模板是一个文本，用于分离文档的表现形式和内容。

模板应用实例

我们接着上一章节的项目将在 HelloWorld 目录底下创建 templates 目录并建立 hello.html文件，整个目录结构如下：

```
HelloWorld/
|-- HelloWorld
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- settings.py
|   |-- settings.pyc
|   |-- urls.py
|   |-- urls.pyc
|   |-- view.py
|   |-- view.pyc
|   |-- wsgi.py
|   `-- wsgi.pyc
|-- manage.py
`-- templates
    `-- hello.html
```

hello.html 文件代码如下：

```
<h1>{{ hello }}</h1>
```

从模板中我们知道变量使用了双括号。

接下来我们需要向Django说明模板文件的路径，修改HelloWorld/settings.py，修改 TEMPLATES 中的 DIRS 为 **[BASE_DIR+"/templates",]**，如下所示:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR+"/templates",],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

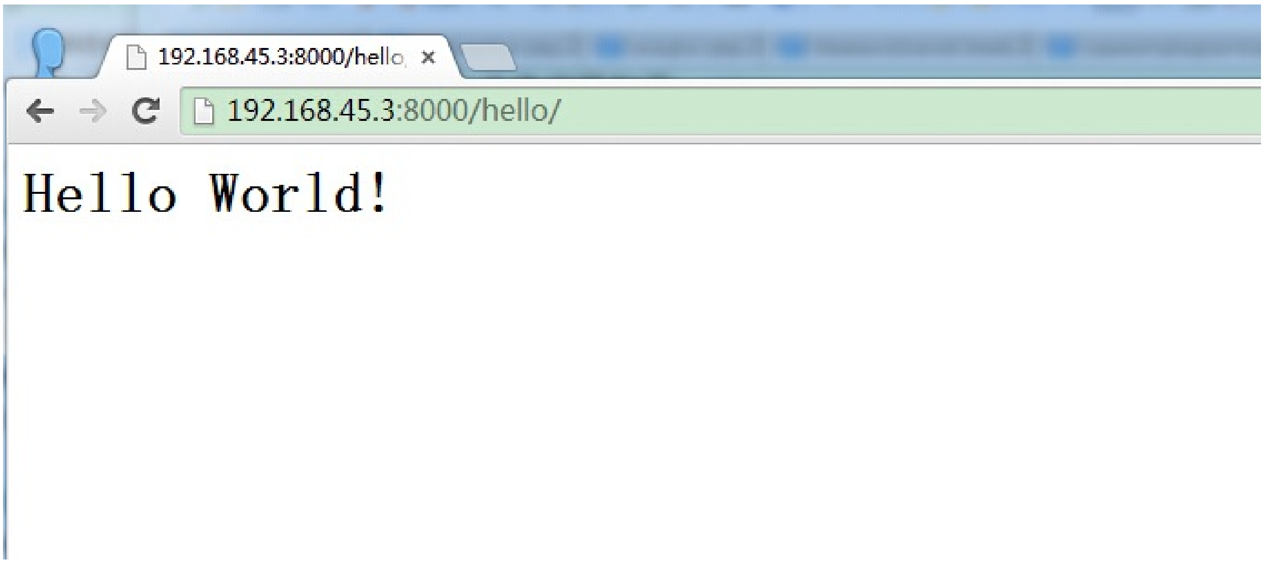
我们现在修改 view.py，增加一个新的对象，用于向模板提交数据：

```
# -*- coding: utf-8 -*-

#from django.http import HttpResponse
from django.shortcuts import render

def hello(request):
    context          = {}
    context['hello'] = 'Hello World!'
    return render(request, 'hello.html', context)
```

可以看到，我们这里使用render来替代之前使用的HttpResponse。render还使用了一个字典context作为参数。

context 字典中元素的键值 "hello" 对应了模板中的变量 "{{ hello }}"。

再访问访问http://192.168.45.3:8000/hello/，可以看到页面：



这样我们就完成了使用模板来输出数据，从而实现数据与视图分离。

接下来我们将具体介绍模板中常用的语法规则。

Django 模板标签

if/else 标签

基本语法格式如下：

```
{% if condition %}
     ... display
{% endif %}
```

或者：

```
{% if condition1 %}
   ... display 1
{% elif condiiton2 %}
   ... display 2
{% else %}
   ... display 3
{% endif %}
```

根据条件判断是否输出。if/else 支持嵌套。

{% if %} 标签接受 and ， or 或者 not 关键字来对多个变量做判断 ， 或者对变量取反（ not)，例如：

```
{% if athlete_list and coach_list %}
     athletes 和 coaches 变量都是可用的。
{% endif %}
```

for 标签

{% for %} 允许我们在一个序列上迭代。

与Python的 for 语句的情形类似，循环语法是 for X in Y ，Y是要迭代的序列而X是在每一个特定的循环中使用的变量名称。

每一次循环中，模板系统会渲染在 {% for %} 和 {% endfor %} 之间的所有内容。

例如，给定一个运动员列表 athlete_list 变量，我们可以使用下面的代码来显示这个列表：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

给标签增加一个 reversed 使得该列表被反向迭代：

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

可以嵌套使用 {% for %} 标签：

```
{% for athlete in athlete_list %}
    <h1>{{ athlete.name }}</h1>
    <ul>
    {% for sport in athlete.sports_played %}
        <li>{{ sport }}</li>
    {% endfor %}
    </ul>
{% endfor %}
```

ifequal/ifnotequal 标签

{% ifequal %} 标签比较两个值，当他们相等时，显示在 {% ifequal %} 和 {% endifequal %} 之中所有的值。

下面的例子比较两个模板变量 user 和 currentuser :

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

和 {% if %} 类似， {% ifequal %} 支持可选的 {% else%} 标签：8

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

注释标签

Django 注释使用 {# #}。

```
{# 这是一个注释 #}
```

过滤器

模板过滤器可以在变量被显示前修改它，过滤器使用管道字符，如下所示：

```
{{ name|lower }}
```

{{ name }} 变量被过滤器 lower 处理后，文档大写转换文本为小写。

过滤管道可以被 套接， 既是说，一个过滤器管道的输出又可以作为下一个管道的输入：

```
{{ my_list|first|upper }}
```

以上实例将第一个元素并将其转化为大写。

有些过滤器有参数。 过滤器的参数跟随冒号之后并且总是以双引号包含。 例如：

```
{{ bio|truncatewords:"30" }}
```

这个将显示变量 bio 的前30个词。

其他过滤器：

- addslashes : 添加反斜杠到任何反斜杠、单引号或者双引号前面。
- date : 按指定的格式字符串参数格式化 date 或者 datetime 对象，实例：

  ```
  {{ pub_date|date:"F j, Y" }}
  ```

- length : 返回变量的长度。

include 标签

{% include %} 标签允许在模板中包含其它的模板的内容。

下面这两个例子都包含了 nav.html 模板：

```
{% include "nav.html" %}
```

模板继承

模板可以用继承的方式来实现复用。

接下来我们先创建之前项目的 templates 目录中添加 base.html 文件，代码如下：

```
<html>
  <head>
    <title>Hello World!</title>
  </head>

  <body>
    <h1>Hello World!</h1>
    {% block mainbody %}
       <p>original</p>
    {% endblock %}
  </body>
</html>
```

以上代码中，名为mainbody的block标签是可以被继承者们替换掉的部分。

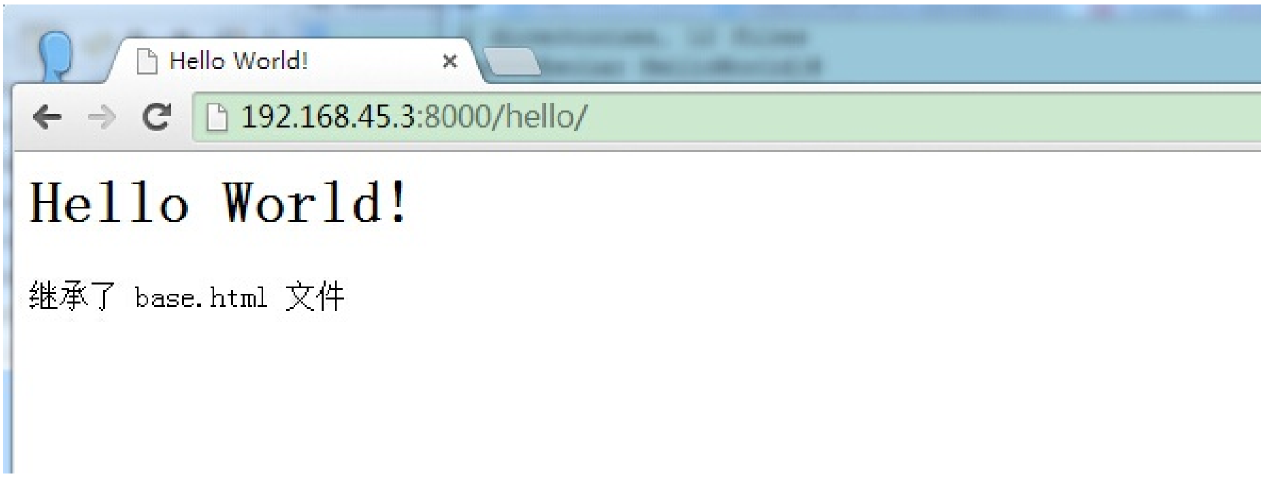
所有的 {% block %} 标签告诉模板引擎，子模板可以重载这些部分。

hello.html中继承base.html，并替换特定block，hello.html修改后的代码如下：

```
{% extends "base.html" %}

{% block mainbody %}
<p>继承了 base.html 文件</p>
{% endblock %}
```

第一行代码说明hello.html继承了 base.html 文件。可以看到，这里相同名字的block标签用以替换base.html的相应block。

重新访问地址http://192.168.45.3:8000/hello/，输出结果如下：



Django 模型

Django 对各种数据库提供了很好的支持，包括：PostgreSQL、MySQL、SQLite、Oracle。

Django 为这些数据库提供了统一的调用API。 我们可以根据自己业务需求选择不同的数据库。

MySQL 是 Web 应用中最常用的数据库。本章节我们将以 Mysql 作为实例进行介绍。你可以通过本站的 MySQL 教程 了解更多Mysql的基础知识。

数据库配置

我们在项目的 settings.py 文件中找到 DATABASES 配置项，将其信息修改为：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'test',
        'USER': 'test',
        'PASSWORD': 'test123',
        'HOST':'localhost',
        'PORT':'3306',
    }
}
```

上面包含数据库名称和用户的信息，它们与MySQL中对应数据库和用户的设置相同。Django根据这一设置，与MySQL中相应的数据库和用户连接起来。

定义模型

创建 APP

Django规定，如果要使用模型，必须要创建一个app。我们使用以下命令创建一个 TestModel 的 app:

```
python manage.py startapp TestModel
```

目录结构如下：

```
HelloWorld
|-- TestModel
|   |-- __init__.py
|   |-- admin.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
```

我们修改 TestModel/models.py文件，代码如下：

```
# models.py
from django.db import models

class Test(models.Model):
    name = models.CharField(max_length=20)
```

以上的类名代表了数据库表名，且继承了models.Model，类里面的字段代表数据表中的字段(name)，数据类型则由CharField（相当于varchar）、DateField（相当于datetime），max_length 参数限定长度。

接下来在settings.py中找到INSTALLED_APPS这一项，如下：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'TestModel',               # 添加此项
)
```

在命令行中运行python manage.py syncdb，看到几行"Creating table…"的字样，你的数据表就创建好了。

```
Creating tables ...
……
Creating table TestModel_test  #我们自定义的表
……
```

表名组成结构为：app名_类名（如：TestModel_test）。

注意：尽管我们没有在models给表设置主键，但是Django会自动添加一个id作为主键。

数据库操作

接下来我们在 HelloWorld 目录中添加 testdb.py 文件，并修改urls.py：

```
from django.conf.urls import *
from HelloWorld.view import hello
from HelloWorld.testdb import testdb

urlpatterns = patterns("",
        ('^hello/$', hello),
        ('^testdb/$', testdb),
)
```

添加数据

添加数据需要先创建对象，然后再执行 save 函数，相当于SQL中的INSERT：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    test1 = Test(name='w3cschool.cc')
    test1.save()
    return HttpResponse("<p>数据添加成功！</p>")
```

访问http://192.168.45.3:8000/testdb/ 就可以看到数据添加成功的提示。

获取数据

Django提供了多种方式来获取数据库的内容，如下代码所示：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    # 初始化
    response = ""
    response1 = ""

    # 通过objects这个模型管理器的all()获得所有数据行，相当于SQL中的SELECT * FROM
    list = Test.objects.all()

    # filter相当于SQL中的WHERE，可设置条件过滤结果
    response2 = Test.objects.filter(id=1)

    # 获取单个对象
    response3 = Test.objects.get(id=1)

    # 限制返回的数据 相当于 SQL 中的 OFFSET 0 LIMIT 2;
    Test.objects.order_by('name')[0:2]

    #数据排序
    Test.objects.order_by("id")

    # 上面的方法可以连锁使用
    Test.objects.filter(name="w3cschool.cc").order_by("id")

    # 输出所有数据
    for var in list:
        response1 += var.name + " "
    response = response1
    return HttpResponse("<p>" + response + "</p>")
```

输出结果如下图所示：

更新数据

修改数据可以使用 save() 或 update():

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    # 修改其中一个id=1的name字段，再save，相当于SQL中的UPDATE
    test1 = Test.objects.get(id=1)
    test1.name = 'w3cschool菜鸟教程'
    test1.save()

    # 另外一种方式
    #Test.objects.filter(id=1).update(name='w3cschool菜鸟教程')

    # 修改所有的列
    # Test.objects.all().update(name='w3cschool菜鸟教程')

    return HttpResponse("<p>修改成功</p>")
```

删除数据

删除数据库中的对象只需调用该对象的delete()方法即可：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    # 删除id=1的数据
    test1 = Test.objects.get(id=1)
    test1.delete()

    # 另外一种方式
    # Test.objects.filter(id=1).delete()

    # 删除所有数据
    # Test.objects.all().delete()

    return HttpResponse("<p>删除成功</p>")
```

Django 表单

HTML表单是网站交互性的经典方式。 本章将介绍如何用Django对用户提交的表单数据进行处理。

HTTP 请求

HTTP协议以"请求－回复"的方式工作。客户发送请求时，可以在请求中附加数据。服务器通过解析请求，就可以获得客户传来的数据，并根据URL来提供特定的服务。

GET 方法

我们在之前的项目中创建一个 search.py 文件，用于接收用户的请求：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse
from django.shortcuts import render_to_response

# 表单
def search_form(request):
    return render_to_response('search_form.html')

# 接收请求数据
def search(request):
    request.encoding='utf-8'
    if 'q' in request.GET:
        message = '你搜索的内容为: ' + request.GET['q'].encode('utf-8')
    else:
        message = '你提交了空表单'
    return HttpResponse(message)
```

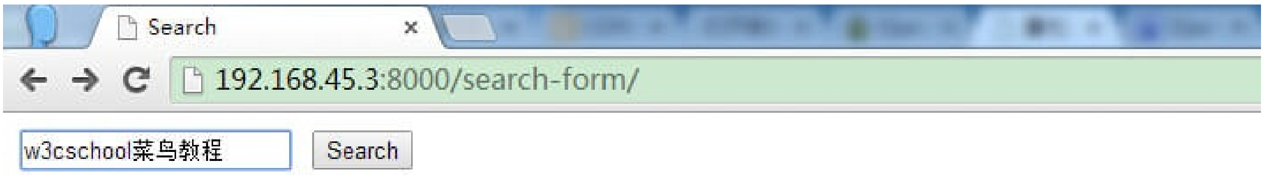
在模板目录template中添加 search_form.html 表单：

```
<html>
<head>
    <meta charset="utf-8" />
    <title>Search - w3cschool.cc</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

urls.py 规则修改为如下形式：

```
from django.conf.urls import *
from HelloWorld.view import hello
from HelloWorld.testdb import testdb
from HelloWorld import search

urlpatterns = patterns("",
    ('^hello/$', hello),
    ('^testdb/$', testdb),
    (r'^search-form/$', search.search_form),
    (r'^search/$', search.search),
)
```

访问地址：http://192.168.45.3:8000/search-form/并搜索，结果如下所示:



POST 方法

上面我们使用了GET方法。视图显示和请求处理分成两个函数处理。

提交数据时更常用POST方法。我们下面使用该方法，并用一个URL和处理函数，同时显示视图和处理请求。

我们在tmplate 创建 post.html：

```
<html>
<head>
    <meta charset="utf-8" />
    <title>Search - w3cschool.cc</title>
</head>
<body>
    <form action="/search-post/" method="post">
        {% csrf_token %}
        <input type="text" name="q">
        <input type="submit" value="Submit">
    </form>

    <p>{{ rlt }}</p>
</body>
</html>
```

在模板的末尾，我们增加一个rlt记号，为表格处理结果预留位置。

表格后面还有一个{% csrf_token %}的标签。csrf全称是Cross Site Request Forgery。这是Django提供的防止伪装提交请求的功能。POST方法提交的表格，必须有此标签。

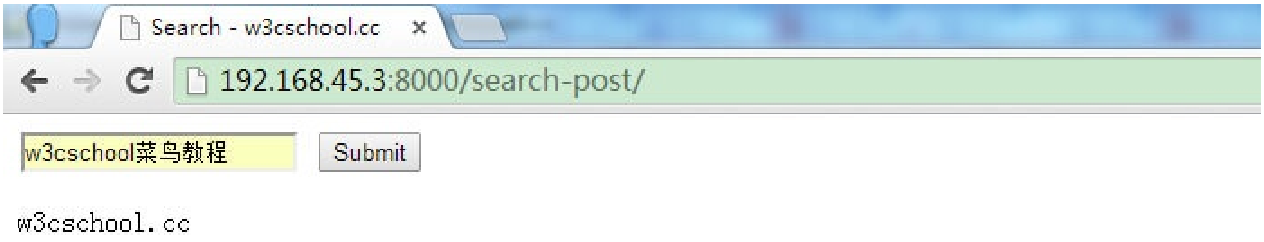
在HelloWorld目录下新建 search2.py 文件并使用 search_post 函数来处理 POST 请求：

```
# -*- coding: utf-8 -*-

from django.shortcuts import render
from django.core.context_processors import csrf

# 接收POST请求数据
def search_post(request):
    ctx ={}
    ctx.update(csrf(request))
    if request.POST:
        ctx['rlt'] = request.POST['q']
    return render(request, "post.html", ctx)
```

urls.py 规则修改为如下形式：

```
from django.conf.urls import *
from HelloWorld.view import hello
from HelloWorld.testdb import testdb
from HelloWorld import search
from HelloWorld import search2

urlpatterns = patterns("",
    ('^hello/$', hello),
    ('^testdb/$', testdb),
    (r'^search-form/$', search.search_form),
    (r'^search/$', search.search),
    (r'^search-post/$', search2.search_post),
)
```

访问 http://192.168.45.3:8000/search-post/ 显示结果如下：



完成以上实例后，我们的目录结构为：

```
HelloWorld
|-- HelloWorld
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- models.pyc
|   |-- search.py
|   |-- search.pyc
|   |-- search2.py
|   |-- search2.pyc
|   |-- settings.py
|   |-- settings.pyc
|   |-- testdb.py
|   |-- testdb.pyc
|   |-- urls.py
|   |-- urls.pyc
|   |-- view.py
|   |-- view.pyc
|   |-- wsgi.py
|   `-- wsgi.pyc
|-- TestModel
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- admin.py
|   |-- models.py
|   |-- models.pyc
|   |-- tests.py
|   `-- views.py
|-- manage.py
`-- templates
    |-- base.html
    |-- hello.html
    |-- post.html
    `-- search_form.html

3 directories, 29 files
```

Request 对象

每个view函数的第一个参数是一个HttpRequest对象，就像下面这个hello()函数:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

HttpRequest对象包含当前请求URL的一些信息：

属性	描述
`path`	请求页面的全路径,不包括域名—例如, "/hello/"。
`method`	请求中使用的HTTP方法的字符串表示。全大写表示。例如: `if request.method == 'GET':` `do_something()` `elif request.method == 'POST':` `do_something_else()`
`GET`	包含所有HTTP GET参数的类字典对象。参见QueryDict 文档。
`POST`	包含所有HTTP POST参数的类字典对象。参见QueryDict 文档。服务器收到空的POST请求的情况也是有可能发生的。也就是说，表单form通过HTTP POST方法提交请求，但是表单中可以没有数据。因此，不能使用语句if request.POST来判断是否使用HTTP POST方法；应该使用if request.method == "POST" (参见本表的method属性)。注意: POST不包括file-upload信息。参见FILES属性。
`REQUEST`	为了方便，该属性是POST和GET属性的集合体，但是有特殊性，先查找POST属性，然后再查找GET属性。借鉴PHP's `$_REQUEST` 。例如，如果 `GET = {"name": "john"}` 和 `POST = {"age": '34'}` ,则 `REQUEST["name"]` 的值是"john", `REQUEST["age"]` 的值是"34".强烈建议使用GET and POST,因为这两个属性更加显式化，写出的代码也更易理解。
`COOKIES`	包含所有cookies的标准Python字典对象。Keys和values都是字符串。参见第12章，有关于cookies更详细的讲解。
`FILES`	包含所有上传文件的类字典对象。FILES中的每个Key都是<input type="file" name="" />标签中name属性的值. FILES中的每个value 同时也是一个标准Python字典对象，包含下面三个Keys: `filename` ：上传文件名,用Python字符串表示 `content-type` ：上传文件的Content type `content` ：上传文件的原始内容 注意：只有在请求方法是POST，并且请求页面中<form>有enctype="multipart/form-data"属性时FILES才拥有数据。否则，FILES 是一个空字典。
`META`	包含所有可用HTTP头部信息的字典。 例如: `CONTENT_LENGTH` `CONTENT_TYPE` `QUERY_STRING` ：未解析的原始查询字符串 `REMOTE_ADDR` ：客户端IP地址 `REMOTE_HOST` ：客户端主机名 `SERVER_NAME` ：服务器主机名 `SERVER_PORT` ：服务器端口 META 中这些头加上前缀HTTP_最为Key, 例如: `HTTP_ACCEPT_ENCODING` `HTTP_ACCEPT_LANGUAGE` `HTTP_HOST` ：客户发送的HTTP主机头信息 `HTTP_REFERER` ：referring页 `HTTP_USER_AGENT` ：客户端的user-agent字符串 `HTTP_X_BENDER` ：X-Bender头信息
`user`	是一个django.contrib.auth.models.User 对象，代表当前登录的用户。如果访问用户当前没有登录，user将被初始化为django.contrib.auth.models.AnonymousUser的实例。你可以通过user的is_authenticated()方法来辨别用户是否登录： `if request.user.is_authenticated():` `# Do something for logged-in users.` `else:` `# Do something for anonymous users.` 只有激活Django中的AuthenticationMiddleware时该属性才可用
	唯一可读写的属性，代表当前会话的字典对象。只有激活Django中的

	session支持时该属性才可用。 参见第12章。
`raw_post_data`	原始HTTP POST数据，未解析过。 高级处理时会有用处。

Request对象也有一些有用的方法：

方法	描述
getitem(key)	返回GET/POST的键值,先取POST,后取GET。如果键不存在抛出KeyError。这是我们可以使用字典语法访问HttpRequest对象。例如,request["foo"]等同于先request.POST["foo"] 然后request.GET["foo"]的操作。
has_key()	检查request.GET or request.POST中是否包含参数指定的Key。
get_full_path()	返回包含查询字符串的请求路径。例如， "/music/bands/the_beatles/?print=true"
is_secure()	如果请求是安全的，返回True，就是说，发出的是HTTPS请求。

QueryDict对象

在HttpRequest对象中, GET和POST属性是django.http.QueryDict类的实例。

QueryDict类似字典的自定义类，用来处理单键对应多值的情况。

QueryDict实现所有标准的词典方法。还包括一些特有的方法：

方法	描述
`__getitem__`	和标准字典的处理有一点不同，就是，如果Key对应多个Value，**getitem**()返回最后一个value。
`__setitem__`	设置参数指定key的value列表(一个Python list)。注意：它只能在一个mutable QueryDict 对象上被调用(就是通过copy()产生的一个QueryDict对象的拷贝).
`get()`	如果key对应多个value，get()返回最后一个value。
`update()`	参数可以是QueryDict，也可以是标准字典。和标准字典的update方法不同，该方法添加字典 items，而不是替换它们: `>>> q = QueryDict('a=1')` `>>> q = q.copy() # to make it mutable` `>>> q.update({'a': '2'})` `>>> q.getlist('a')` `['1', '2']` `>>> q['a'] # returns the last` `['2']`
`items()`	和标准字典的items()方法有一点不同,该方法使用单值逻辑的**getitem**(): `>>> q = QueryDict('a=1&a=2&a=3')` `>>> q.items()` `[('a', '3')]`
`values()`	和标准字典的values()方法有一点不同,该方法使用单值逻辑的**getitem**():

此外, QueryDict也有一些方法，如下表：

方法	描述
`copy()`	返回对象的拷贝，内部实现是用Python标准库的copy.deepcopy()。该拷贝是mutable(可更改的) — 就是说，可以更改该拷贝的值。
`getlist(key)`	返回和参数key对应的所有值，作为一个Python list返回。如果key不存在，则返回空list。 It's guaranteed to return a list of some sort..
`setlist(key,list_)`	设置key的值为 `list_ (unlike __setitem__())` .
`appendlist(key,item)`	添加item到和key关联的内部list.
`setlistdefault(key,list)`	和setdefault有一点不同，它接受list而不是单个value作为参数。
`lists()`	和items()有一点不同, 它会返回key的所有值，作为一个list, 例如: `>>> q = QueryDict('a=1&a=2&a=3')` `>>> q.lists()` `[('a', ['1', '2', '3'])]`
`urlencode()`	返回一个以查询字符串格式进行格式化后的字符串(e.g., "a=2&b=3&b=5").

Django Admin 管理工具

Django 提供了基于 web 的管理工具。

Django自动管理工具是django.contrib的一部分。你可以在项目的 settings.py 中的 INSTALLED_APPS看到它：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```

django.contrib是一套庞大的功能集，它是Django基本代码的组成部分。

激活管理工具

通常我们在生成项目时会在urls.py中自动设置好，我们只需去掉注释即可。

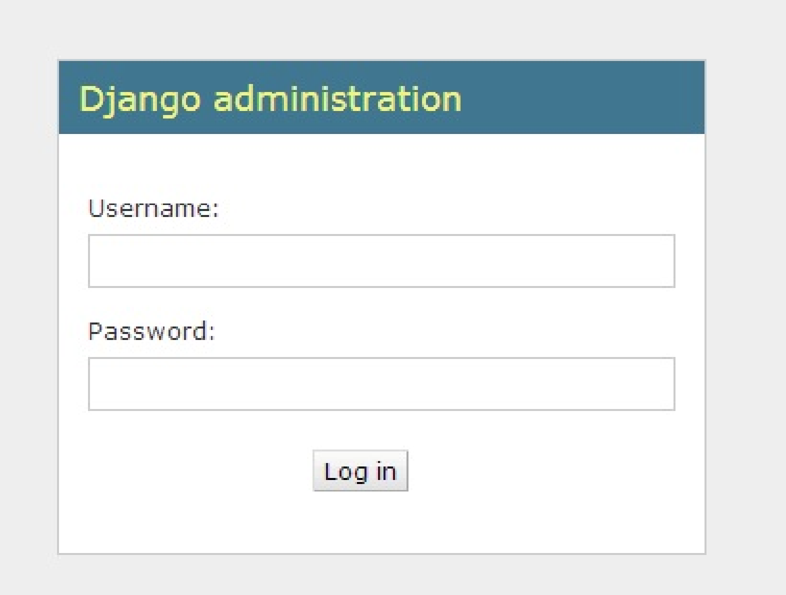
配置项如下所示：

```
from django.contrib import admin
admin.autodiscover()

# And include this URLpattern...
urlpatterns = patterns('',
    # ...
    (r'^admin/', include(admin.site.urls)),
    # ...
)
```

当这一切都配置好后，Django管理工具就可以运行了。

使用管理工具

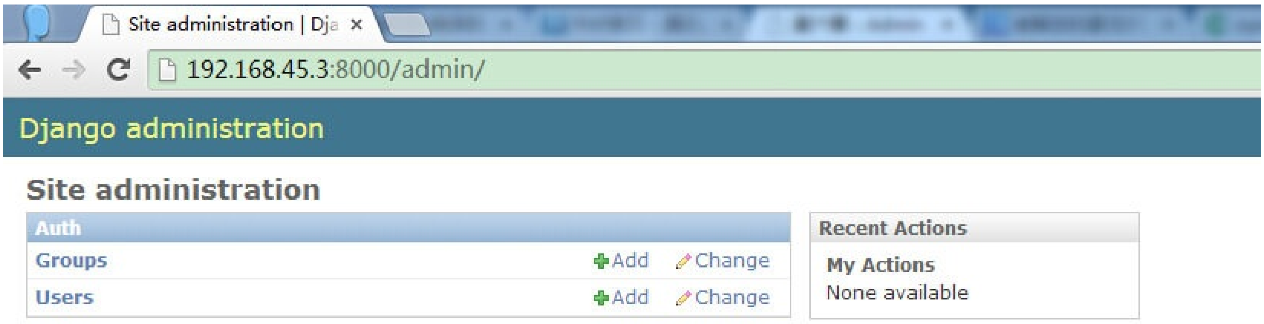
启动开发服务器，然后在浏览器中访问：http://yoursite:8000/admin/，得到如下界面：



你可以通过命令 **python manage.py createsuperuser** 来创建超级用户，如下所示：

```
# python manage.py createsuperuser
Username (leave blank to use 'root'): admin
Email address: admin@w3cschool.cc
Password:
Password (again):
Superuser created successfully.
[root@solar HelloWorld]#
```

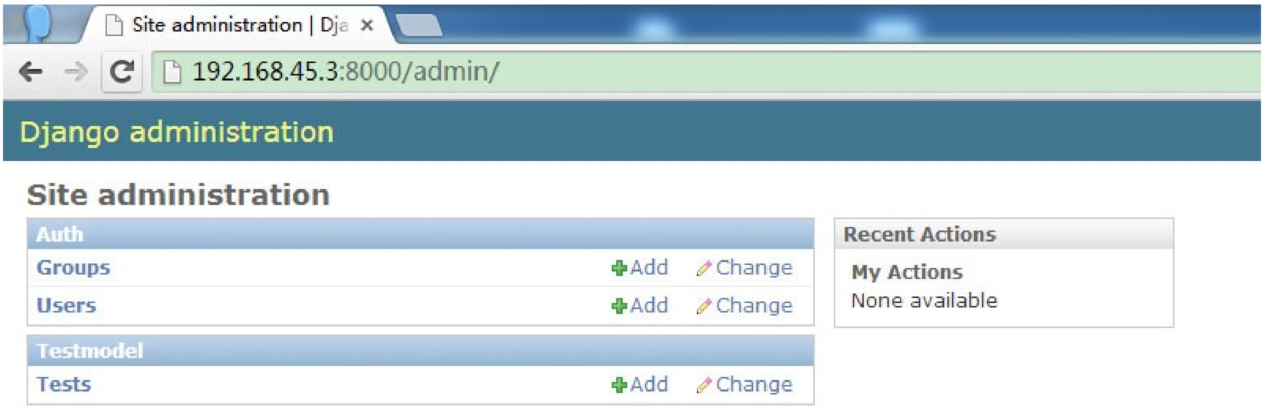
之后输入用户名密码登录，界面如下：



为了让admin界面管理某个数据模型，我们需要先注册该数据模型到admin。比如，我们之前在 TestModel 中已经创建了模型 Test 。修改 TestModel/admin.py:

```
from django.contrib import admin
from TestModel.models import Test

# Register your models here.
admin.site.register(Test)
```

刷新后即可看到 Testmodel 数据表:



复杂模型

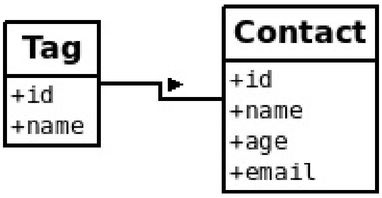
管理页面的功能强大，完全有能力处理更加复杂的数据模型。

先在 TestModel/models.py 中增加一个更复杂的数据模型：

```
from django.db import models

# Create your models here.
class Contact(models.Model):
    name    = models.CharField(max_length=200)
    age     = models.IntegerField(default=0)
    email  = models.EmailField()
    def __unicode__(self):
        return self.name

class Tag(models.Model):
    contact = models.ForeignKey(Contact)
    name    = models.CharField(max_length=50)
    def __unicode__(self):
        return self.name
```

这里有两个表。Tag以Contact为外部键。一个Contact可以对应多个Tag。

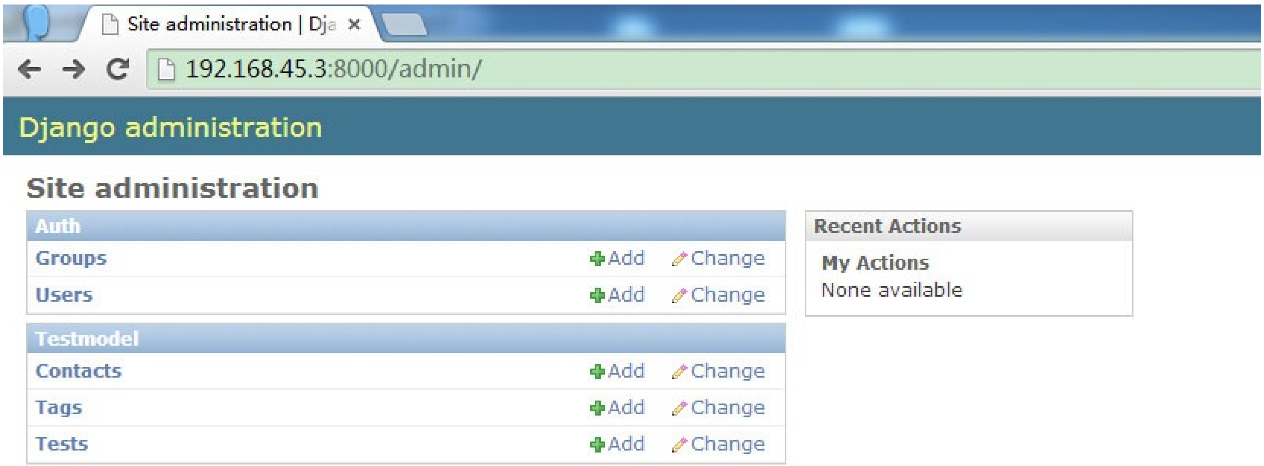
我们还可以看到许多在之前没有见过的属性类型，比如IntegerField用于存储整数。



在 TestModel/admin.py 注册多个模型并显示：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
admin.site.register([Test, Contact, Tag])
```

刷新管理页面，显示结果如下：



在以上管理工具我们就能进行复杂模型操作。

自定义表单

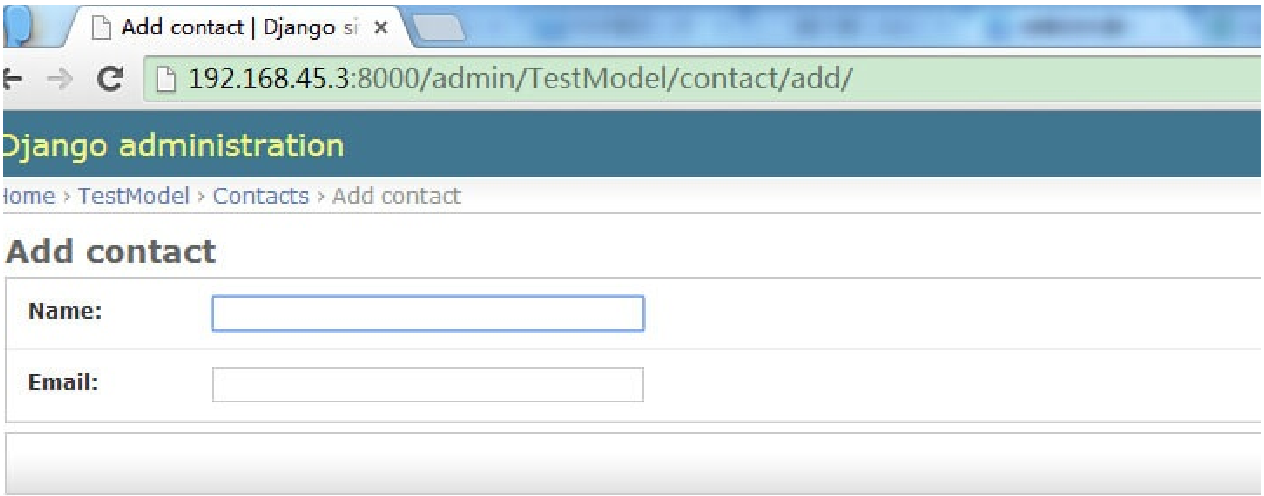
我们可以自定义管理页面，来取代默认的页面。比如上面的"add"页面。我们想只显示name和email部分。修改 TestModel/admin.py:

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    fields = ('name', 'email')

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test, Tag])
```

以上代码定义了一个ContactAdmin类，用以说明管理页面的显示格式。

里面的fields属性定义了要显示的字段。

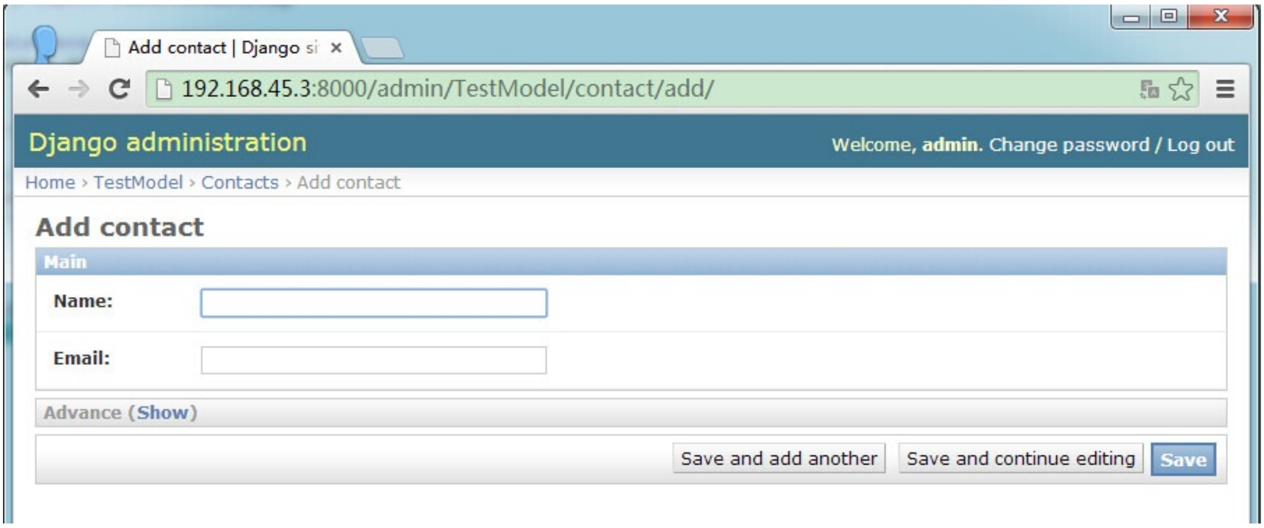
由于该类对应的是Contact数据模型，我们在注册的时候，需要将它们一起注册。显示效果如下：



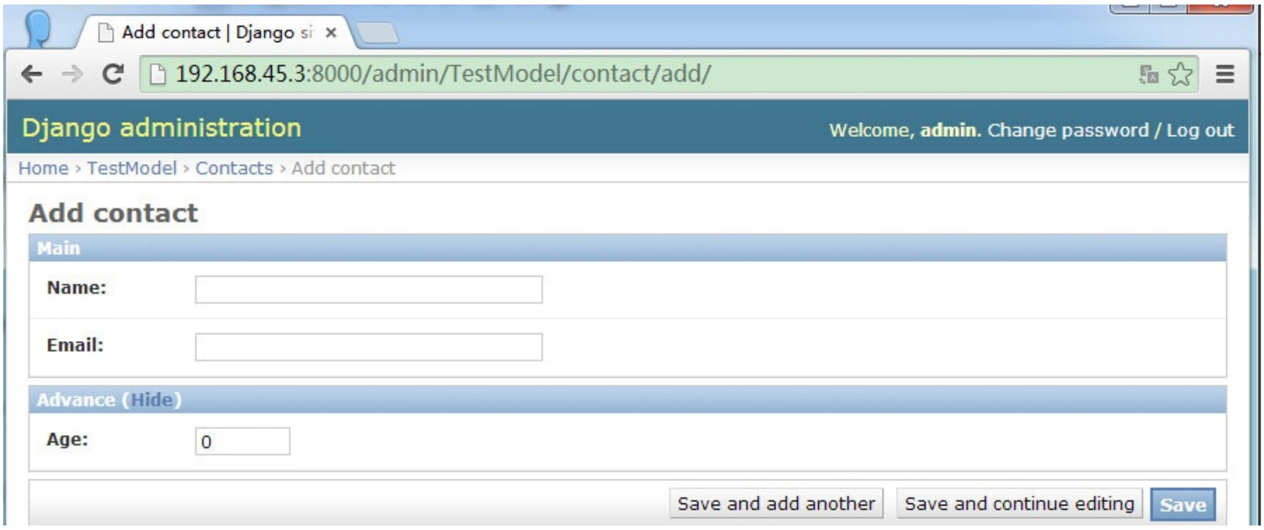
我们还可以将输入栏分块，每个栏也可以定义自己的格式。修改TestModel/admin.py为：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',), # CSS
            'fields': ('age',),
        }]
    )

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test, Tag])
```

上面的栏目分为了Main和Advance两部分。classes说明它所在的部分的CSS格式。这里让Advance部分隐藏：



Advance部分旁边有一个 Show 按钮，用于展开，展开后可点击 Hide 将其隐藏，如下图所示：



内联(Inline)显示

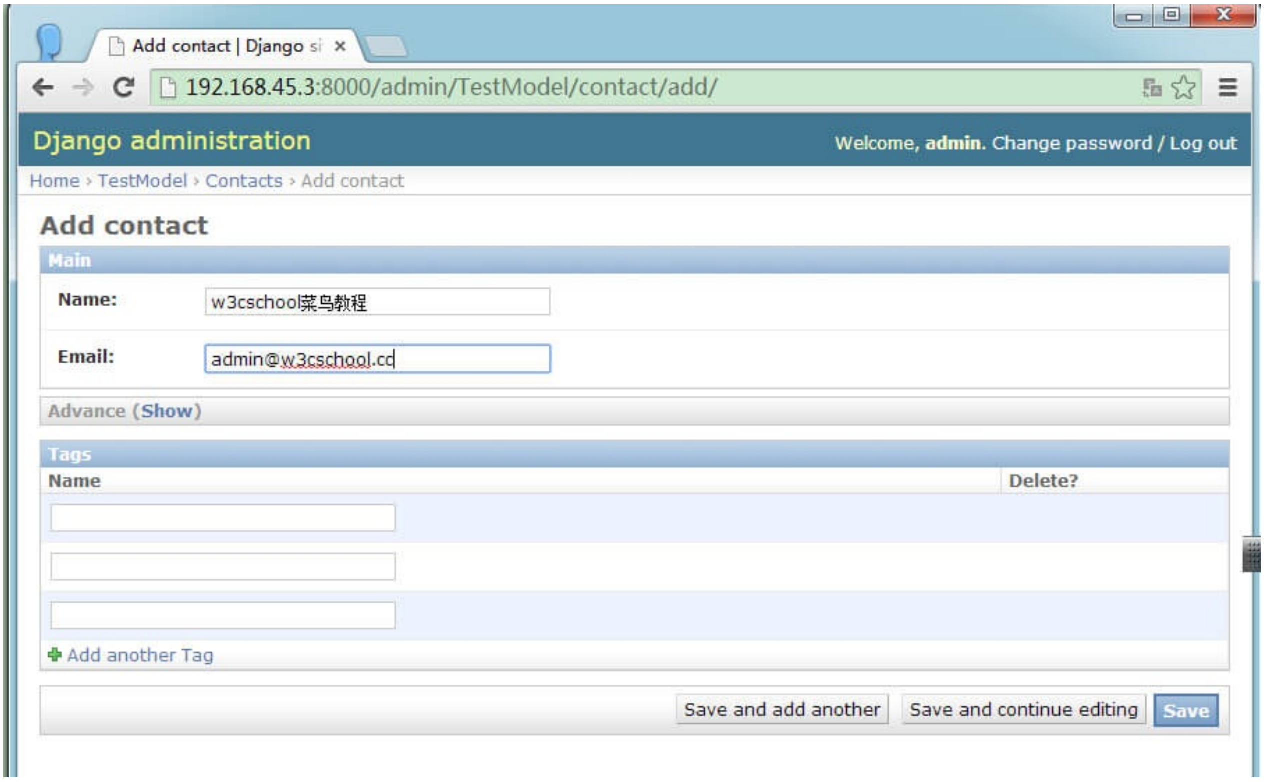
上面的Contact是Tag的外部键，所以有外部参考的关系。

而在默认的页面显示中，将两者分离开来，无法体现出两者的从属关系。我们可以使用内联显示，让Tag附加在Contact的编辑页面上显示。

修改TestModel/admin.py：

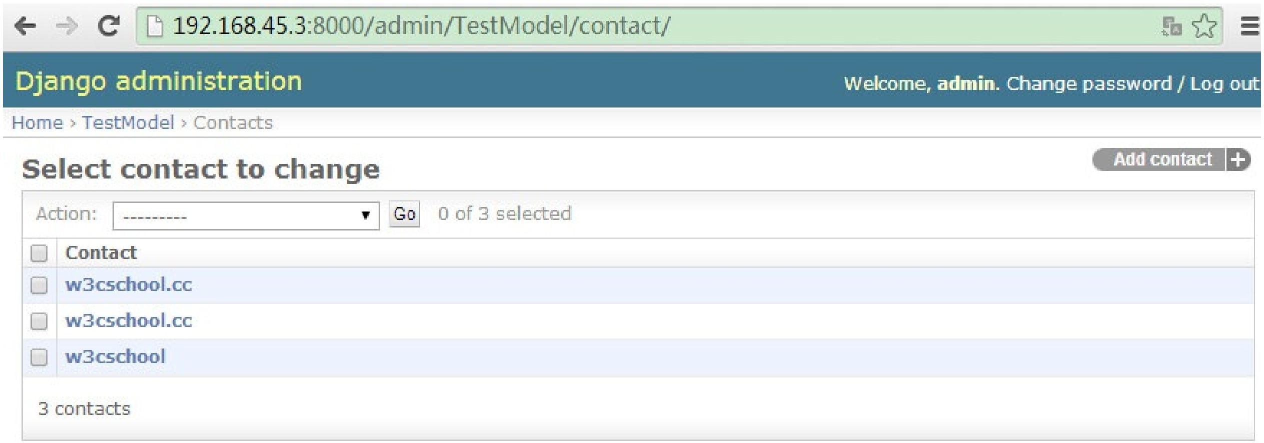
```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class TagInline(admin.TabularInline):
    model = Tag

class ContactAdmin(admin.ModelAdmin):
    inlines = [TagInline]  # Inline
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',),
            'fields': ('age',),
        }]

    )

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test])
```

显示效果如下：



列表页的显示

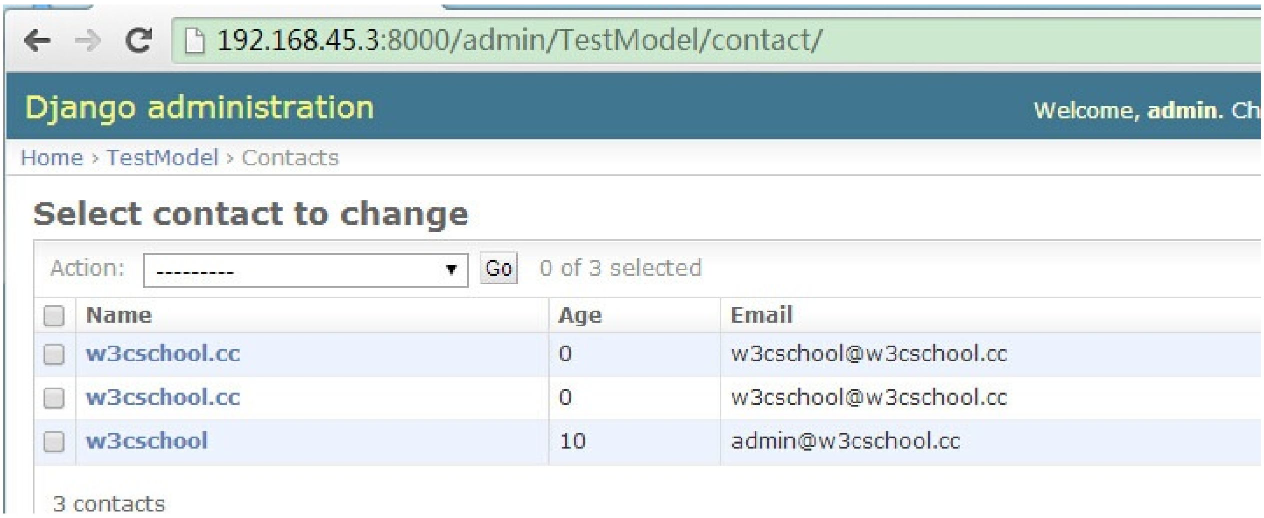
在Contact输入数条记录后，Contact的列表页看起来如下:



我们也可以自定义该页面的显示，比如在列表中显示更多的栏目，只需要在ContactAdmin中增加list_display属性:

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    list_display = ('name','age', 'email') # list

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test, Tag])
```

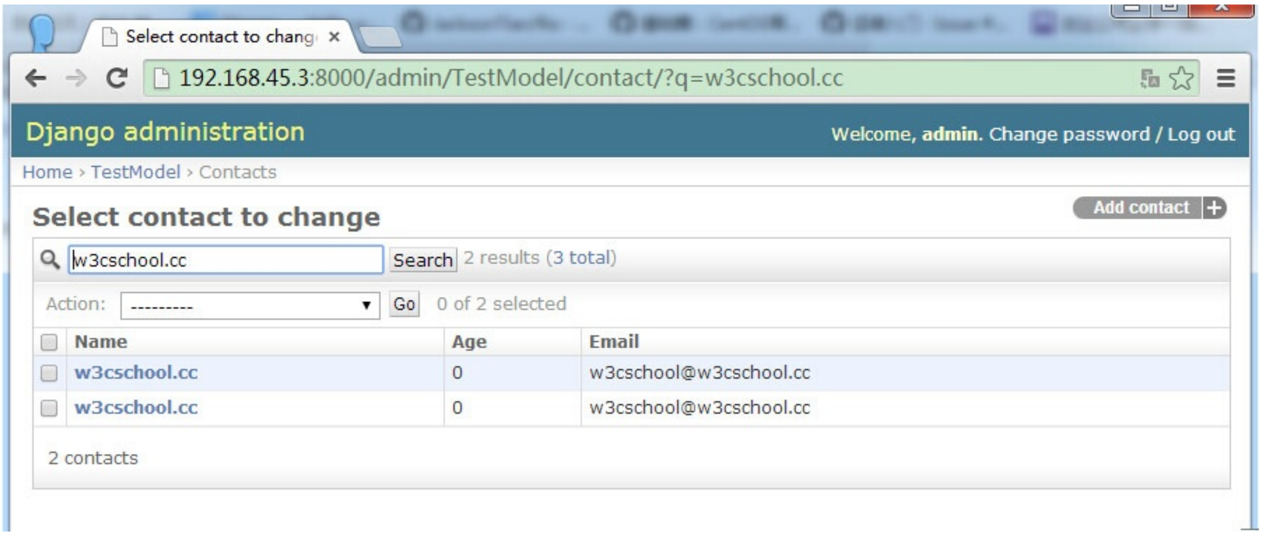
刷新页面显示效果如下：



搜索功能在管理大量记录时非常有，我们可以使用search_fields为该列表页增加搜索栏：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    list_display = ('name','age', 'email')
    search_fields = ('name',)

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test])
```

在本实例中我们搜索了 name 为 w3cschool.cc(本站域名) 的记录，显示结果如下：



Django Admin 管理工具还有非常多实用的功能，感兴趣的同学可以深入研究下。

Django Nginx+uwsgi 安装配置

在前面的章节中我们使用 **python manage.py runserver** 来运行服务器。这只适用测试环境中使用。

正式发布的服务，我们需要一个可以稳定而持续的服务器，比如apache, Nginx, lighttpd等，本文将以 Nginx 为例。

安装基础开发包

Centos 下安装步骤如下：

```
yum groupinstall "Development tools"
yum install zlib-devel bzip2-devel pcre-devel openssl-devel ncurses-devel sqlite-devel re
```

CentOS 自带 Python 2.4.3，但我们可以再安装Python2.7.5：

```
cd ~
wget http://python.org/ftp/python/2.7.5/Python-2.7.5.tar.bz2
tar xvf Python-2.7.5.tar.bz2
cd Python-2.7.5
./configure --prefix=/usr/local
make && make altinstall
```

安装Python包管理

easy_install包 https://pypi.python.org/pypi/distribute

安装步骤:

```
cd ~
wget https://pypi.python.org/packages/source/d/distribute/distribute-0.6.49.tar.gz
tar xf distribute-0.6.49.tar.gz
cd distribute-0.6.49
python2.7 setup.py install
easy_install --version
```

pip包: https://pypi.python.org/pypi/pip

安装pip的好处是可以 pip list、pip uninstall 管理Python包， easy_install 没有这个功能，只有uninstall

安装uwsgi

uwsgi:https://pypi.python.org/pypi/uWSGI

uwsgi参数详解：http://uwsgi-docs.readthedocs.org/en/latest/Options.html

```
pip install uwsgi
uwsgi --version    #查看 uwsgi 版本
```

测试uwsgi是否正常：

新建test.py文件，内容如下：

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return "Hello World"
```

然后在终端运行：

```
uwsgi --http :8001 --wsgi-file test.py
```

在浏览器内输入：http://127.0.0.1:8001，查看是否有"Hello World"输出，若没有输出，请检查你的安装过程。

安装 Django

```
pip install django
```

测试django是否正常，运行：

```
django-admin.py startproject demosite
cd demosite
python2.7 manage.py runserver 0.0.0.0:8002
```

在浏览器内输入：http://127.0.0.1:8002，检查django是否运行正常。

安装 Nginx

安装命令如下：

```
cd ~
wget http://nginx.org/download/nginx-1.5.6.tar.gz
tar xf nginx-1.5.6.tar.gz
cd nginx-1.5.6
./configure --prefix=/usr/local/nginx-1.5.6 \
--with-http_stub_status_module \
--with-http_gzip_static_module
make && make install
```

你可以阅读 Nginx 安装配置 了解更多内容。

uwsgi 配置

uwsgi支持ini、xml等多种配置方式，本文以 ini 为例， 在/ect/目录下新建uwsgi9090.ini，添加如下配置：

```
[uwsgi]
socket = 127.0.0.1:9090
master = true         //主进程
vhost = true          //多站模式
no-site = true        //多站模式时不设置入口模块和文件
workers = 2           //子进程数
reload-mercy = 10
vacuum = true         //退出、重启时清理文件
max-requests = 1000
limit-as = 512
buffer-size = 30000
pidfile = /var/run/uwsgi9090.pid    //pid文件，用于下面的脚本启动、停止该进程
daemonize = /website/uwsgi9090.log
```

Nginx 配置

找到nginx的安装目录（如：/usr/local/nginx/），打开conf/nginx.conf文件，修改server配置：

```
server {
        listen       80;
        server_name  localhost;

        location / {
            include  uwsgi_params;
            uwsgi_pass  127.0.0.1:9090;              //必须和uwsgi中的设置一致
            uwsgi_param UWSGI_SCRIPT demosite.wsgi;  //入口文件，即wsgi.py相对于项目根目录的位
            uwsgi_param UWSGI_CHDIR /demosite;       //项目根目录
            index  index.html index.htm;
            client_max_body_size 35m;
        }
    }
```

你可以阅读 Nginx 安装配置 了解更多内容。

设置完成后，在终端运行：

```
uwsgi --ini /etc/uwsgi9090.ini &
/usr/local/nginx/sbin/nginx
```

在浏览器输入：http://127.0.0.1，你就可以看到django的"It work"了。

W3School Node.js教程

来源：Node.js教程

整理：飞龙

Node.js 简介

简单的说 Node.js 就是运行在服务端的 JavaScript。

Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台。

Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎，V8引擎执行Javascript的速度非常快，性能非常好。

谁适合阅读本教程？

如果你是一个前端程序员，你不懂的像PHP、Python或Java等动态编程语言，然后你想创建自己的服务，那么Node.js是一个非常好的选择。

Node.js 是运行在服务端的 JavaScript，如果你熟悉Javascript，那么你将会很容易的学会Node.js。

当然，如果你是后端程序员，想部署一些高性能的服务，那么学习Node.js也是一个非常好的选择。

学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过Javascript,PHP，Java等编程语言，将有助于你更快的了解Node.js编程。

第一个**Node.js**程序：**Hello World**！

脚本模式

以下是我们的第一个Node.js程序：

```
console.log("Hello World");
```

保存该文件，文件名为 helloworld.js， 并通过 node命令来执行：

```
node helloworld.js
```

程序执行后，正常的话，就会在终端输出 Hello World。

交互模式

打开终端，键入node进入命令交互模式，可以输入一条代码语句后立即执行并显示结果，例如：

```
$ node
> console.log('Hello World!');
Hello World!
```

Node.js 安装配置

本章节我们将向大家介绍在window和Linux上安装Node.js的方法。

本安装教程以Node.js v0.10.26 版本为例。

Node.js安装包及源码下载地址为：http://www.nodejs.org/download/。

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.10.26.tar.gz	

Note: Python 2.6 or 2.7 is required to build from source tarballs.

根据不同平台系统选择你需要的Node.js安装包。

注意：Linux上安装Node.js需要安装Python 2.6 或 2.7， 不建议安装Python 3.0以上版本。

Windowv 上安装Node.js

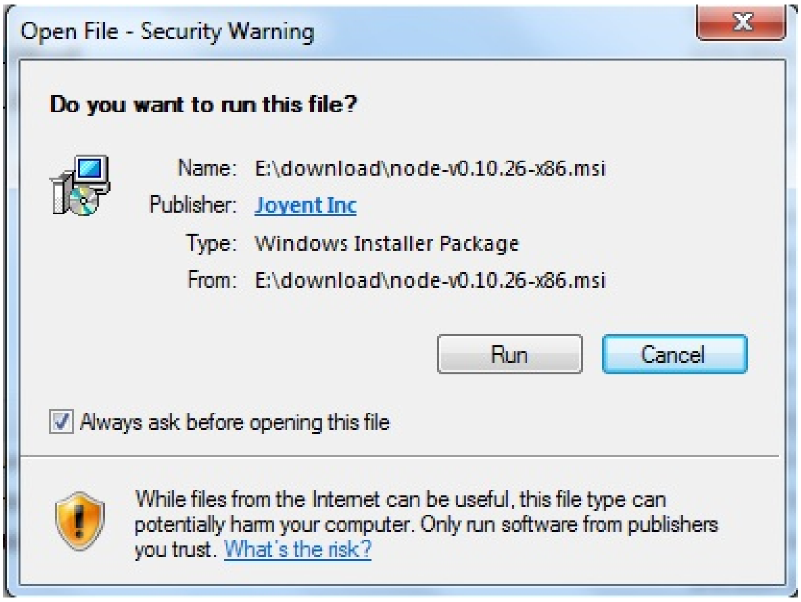
Windows 安装包(.msi)：

32 位安装包下载地址：http://nodejs.org/dist/v0.10.26/node-v0.10.26-x86.msi

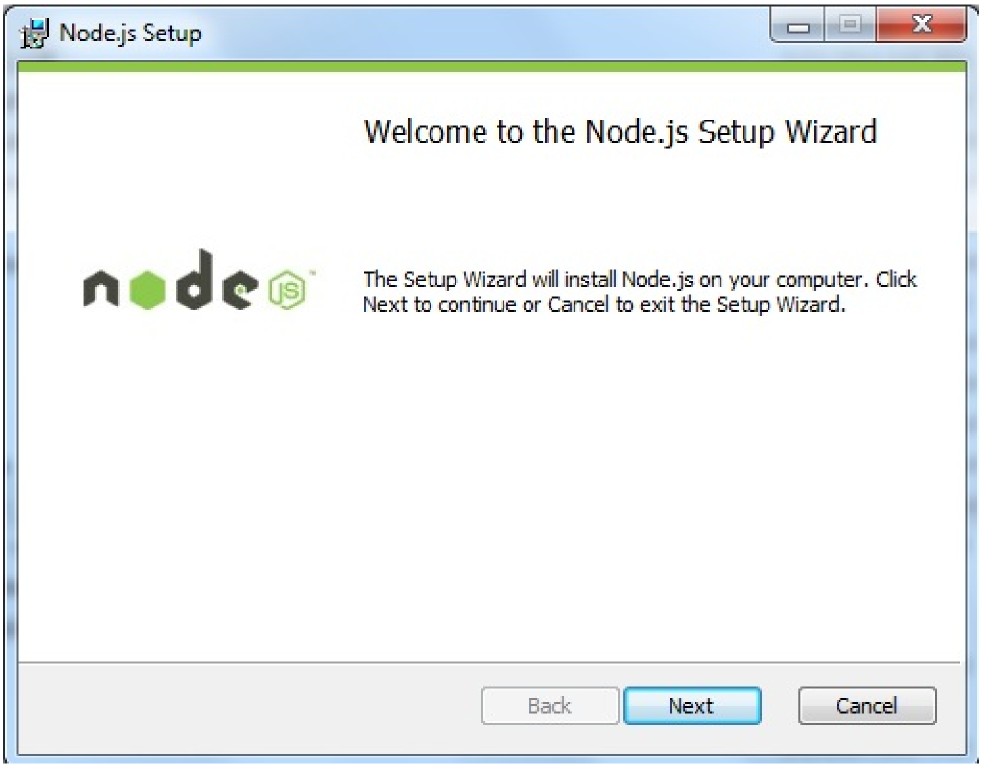
64 位安装包下载地址：http://nodejs.org/dist/v0.10.26/x64/node-v0.10.26-x64.msi

安装步骤：

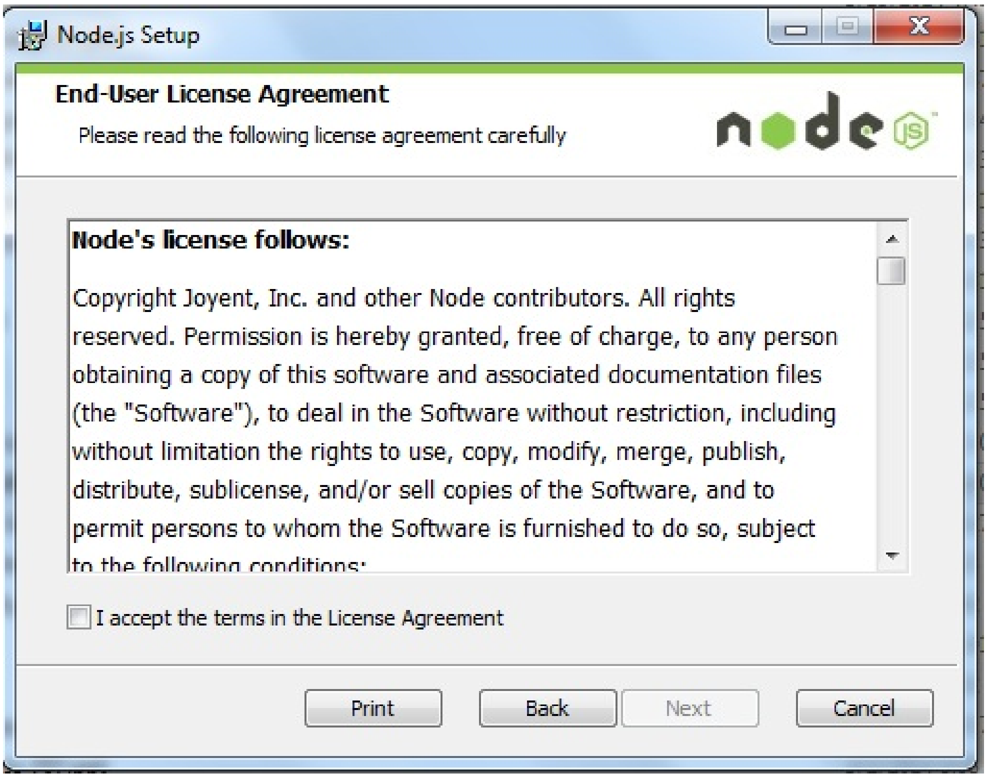
步骤 1：双击下载后的安装包 node-v0.10.26-x86.msi，如下所示：



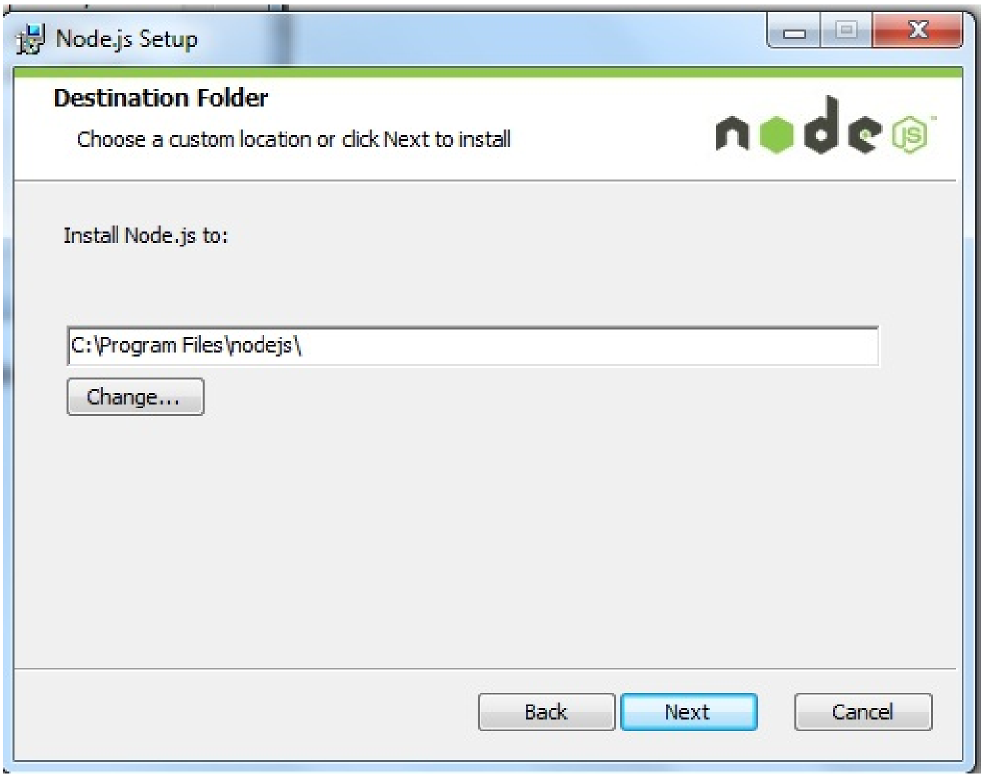
步骤 2 : 点击以上的Run(运行)，将出现如下界面：



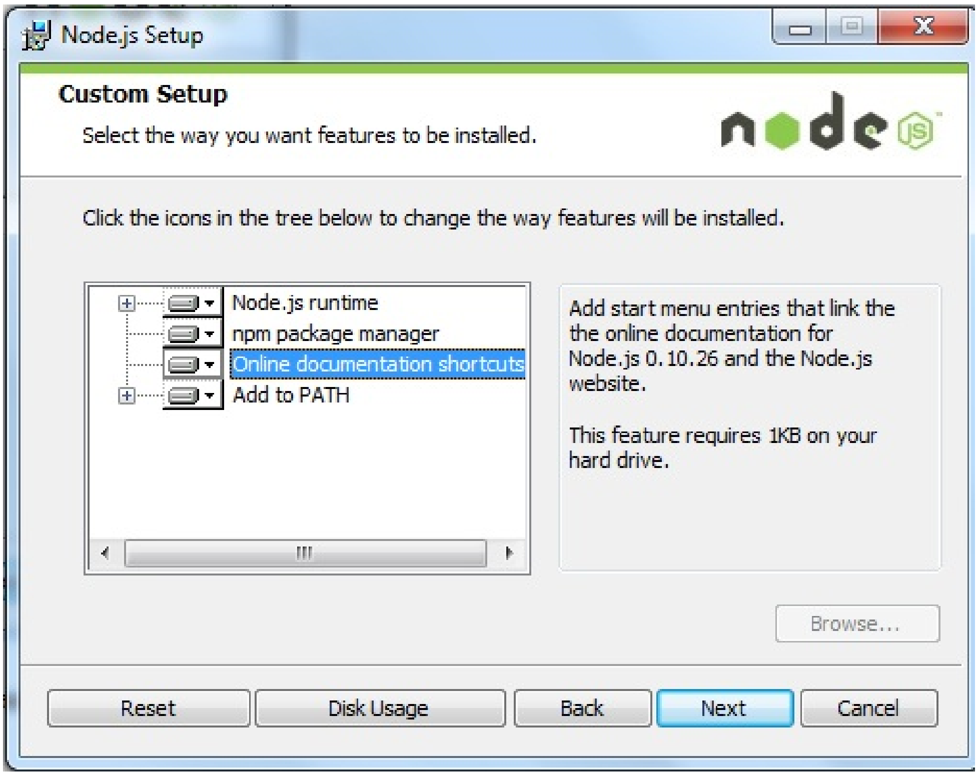
步骤 3 : 勾选接受协议选项，点击 next（下一步） 按钮 :



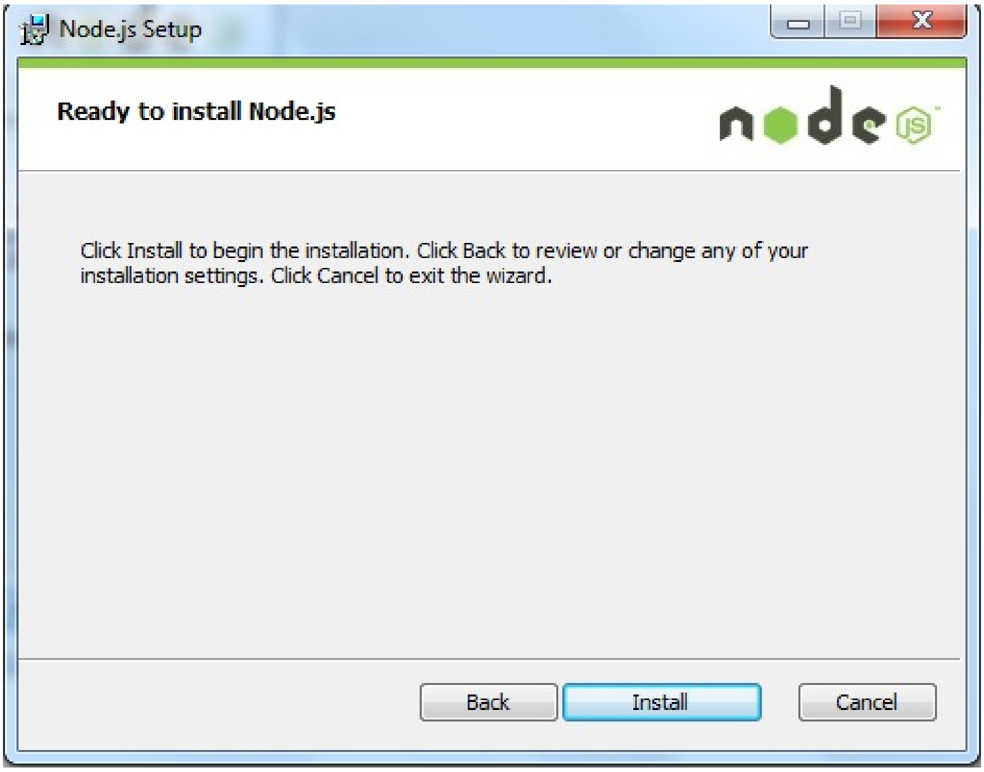
步骤 4 : Node.js默认安装目录为 "C:\Program Files\nodejs\" , 你可以修改目录， 并点击 next（下一步） ：



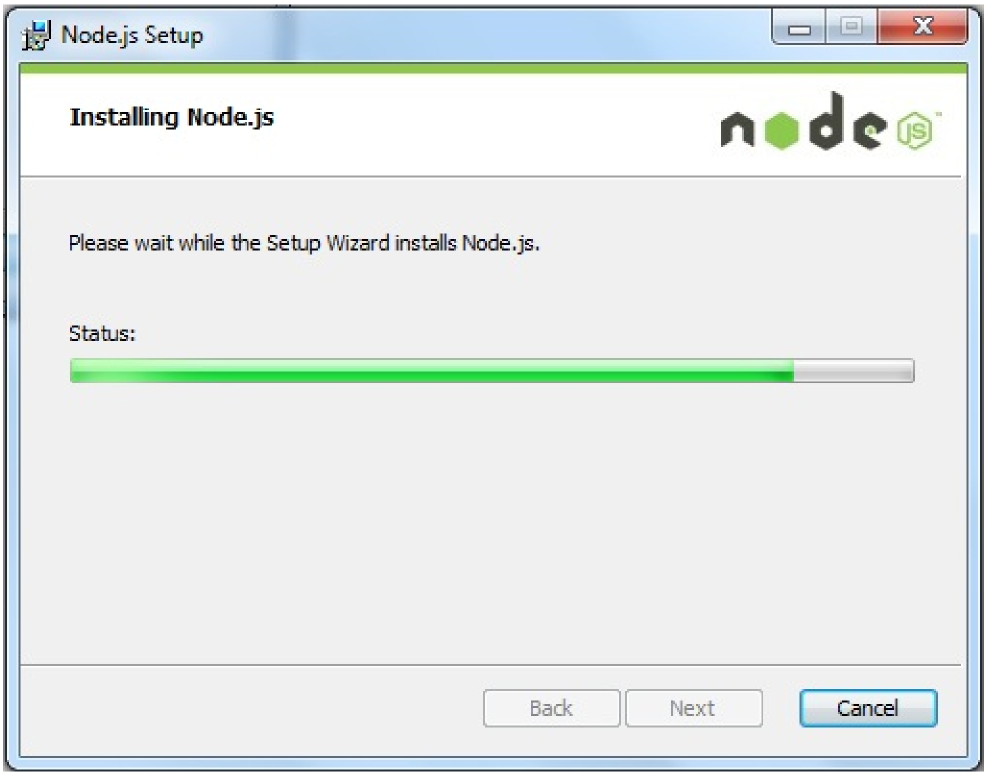
步骤 5 : 点击树形图标来选择你需要的安装模式 , 然后点击下一步 next（下一步）



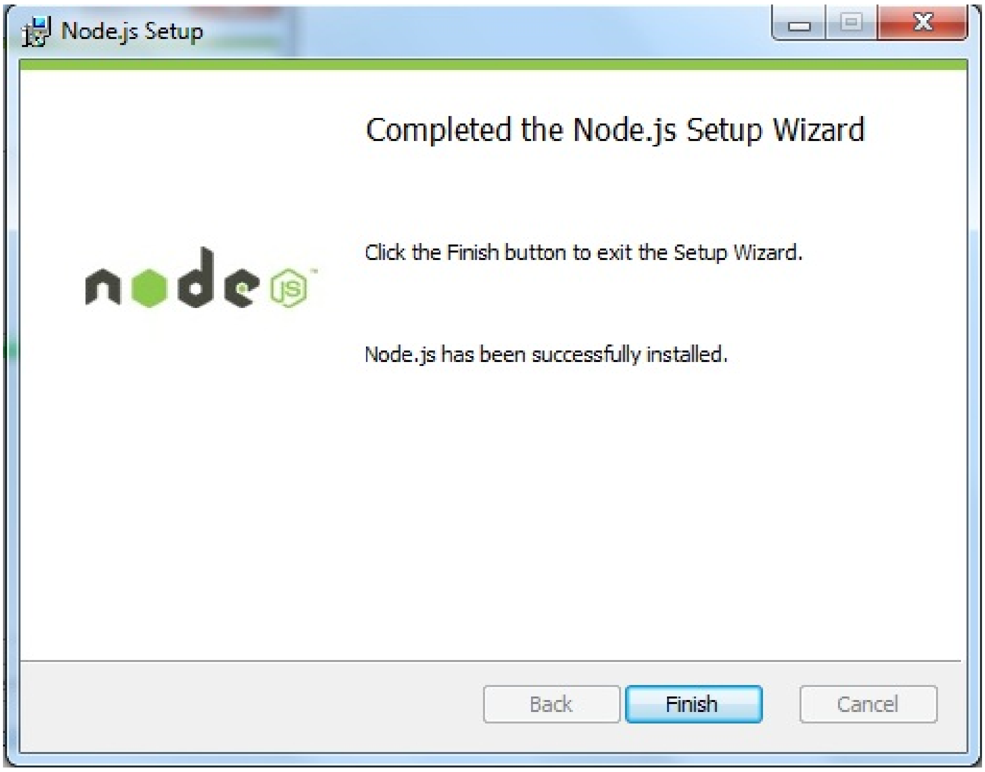
步骤 6 :点击 Install（安装） 开始安装Node.js。你也可以点击 Back（返回）来修改先前的配置。 然后并点击 next（下一步） ：



安装过程：



点击 Finish（完成）按钮退出安装向导。



检测PATH环境变量是否配置了Node.js，点击开始=》运行=》输入"cmd" => 输入命令"path"，输出如下结果：

```
PATH=C:\oraclexe\app\oracle\product\10.2.0\server\bin;C:\Windows\system32;
C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;
c:\python32\python;C:\MinGW\bin;C:\Program Files\GTK2-Runtime\lib;
C:\Program Files\MySQL\MySQL Server 5.5\bin;C:\Program Files\nodejs\;
C:\Users\rg\AppData\Roaming\npm
```

我们可以看到环境变量中已经包含了C:\Program Files\nodejs\

检查Node.js版本

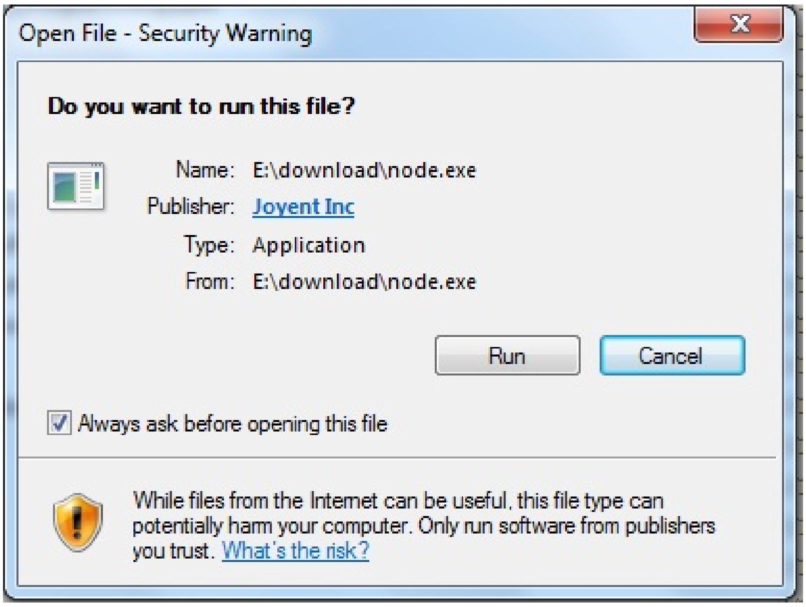
Windows 二进制文件 (.exe)安装：

32 位安装包下载地址：http://nodejs.org/dist/v0.10.26/node.exe

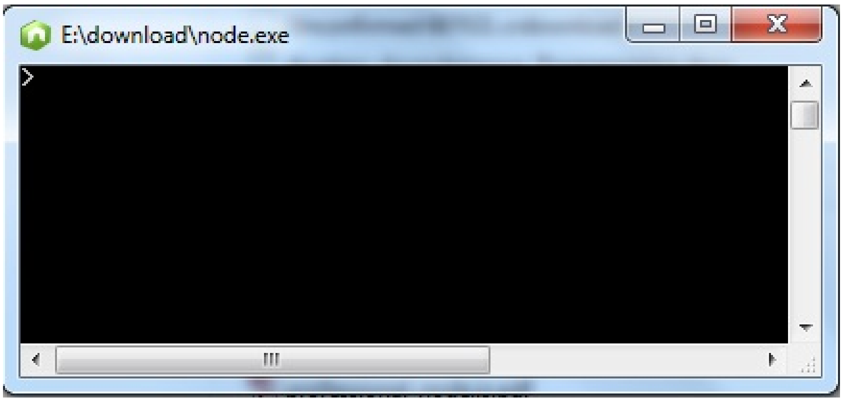
64 位安装包下载地址：http://nodejs.org/dist/v0.10.26/x64/node.exe

安装步骤

步骤 1：双击下载的安装包 Node.exe， 将出现如下界面：



点击 Run（运行）按钮将出现命令行窗口：



版本测试

进入 node.exe 所在的目录，如下所示：

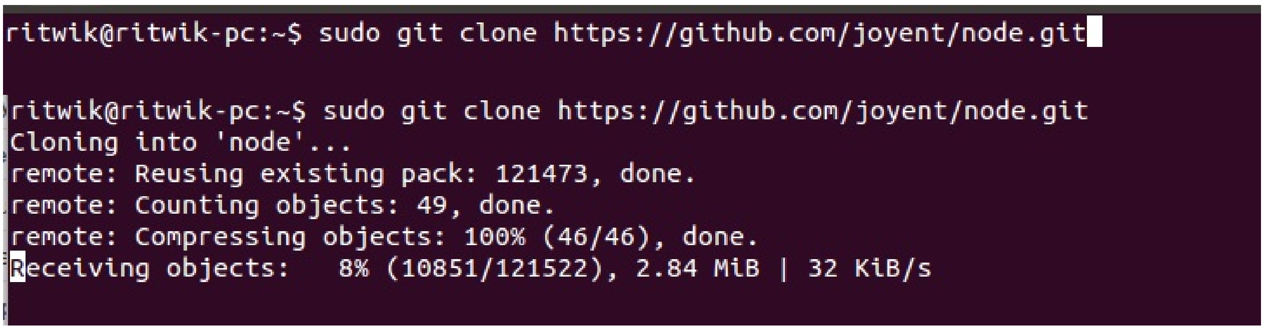
如果你获得以上输出结果，说明你已经成功安装了Node.js。

Linux上安装 Node.js

Ubuntu 安装

以下部分我们将介绍在Ubuntu Linux下安装 Node.js 。 其他的Linux系统，如Centos等类似如下安装步骤。

在 Github 上获取 Node.js 源码：



在完成下载后，将源码包名改为 'node'。

```
ritwik@ritwik-pc:~$ sudo git clone https://github.com/joyent/node.git
Cloning into 'node'...
remote: Reusing existing pack: 121473, done.
remote: Counting objects: 49, done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 121522 (delta 13), reused 4 (delta 3)
Receiving objects: 100% (121522/121522), 91.16 MiB | 34 KiB/s, done.
Resolving deltas: 100% (91075/91075), done.
Checking out files: 100% (9604/9604), done.
ritwik@ritwik-pc:~$
```

修改目录权限：

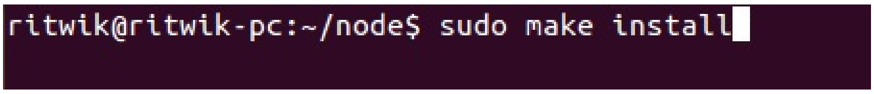
```
ritwik@ritwik-pc:~$ sudo chmod 755 -R node
```

使用 './configure' 创建编译文件。

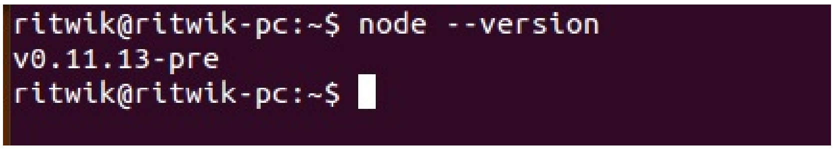
```
ritwik@ritwik-pc:~/node$ sudo ./configure
{ 'target_defaults': { 'cflags': [],
                       'default_configuration': 'Release',
                       'defines': ['OPENSSL_NO_SSL2=1'],
                       'include_dirs': [],
                       'libraries': []},
  'variables': { 'clang': 0,
                 'gcc_version': 47,
                 'host_arch': 'ia32',
                 'node_install_npm': 'true',
                 'node_prefix': '',
                 'node_shared_cares': 'false',
                 'node_shared_http_parser': 'false',
                 'node_shared_libuv': 'false',
                 'node_shared_openssl': 'false',
                 'node_shared_v8': 'false',
                 'node_shared_zlib': 'false',
                 'node_tag': '',
                 'node_use_dtrace': 'false',
                 'node_use_etw': 'false',
                 'node_use_mdb': 'false',
                 'node_use_openssl': 'true',
                 'node_use_perfctr': 'false',
                 'node_v8_options': '',
                 'python': '/usr/bin/python',
                 'target_arch': 'ia32',
                 'uv_library': 'static_library',
                 'uv_parent_path': '/deps/uv/',
                 'uv_use_dtrace': 'false',
                 'v8_enable_gdbjit': 0,
                 'v8_enable_i18n_support': 0,
                 'v8_no_strict_aliasing': 1,
                 'v8_optimized_debug': 0,
                 'v8_random_seed': 0,
                 'v8_use_snapshot': 'true'}}
creating  ./config.gypi
creating  ./config.mk
ritwik@ritwik-pc:~/node$
```

编译: make。

完成安装: make install。



最后我们输入'node --version' 命令来查看Node.js是否安装成功。



centOS下安装nodejs

1、下载源码，你需要在http://nodejs.org/下载最新的Nodejs版本，本文以v0.10.24为例:

```
cd /usr/local/src/
wget http://nodejs.org/dist/v0.10.24/node-v0.10.24.tar.gz
```

2、解压源码

```
tar zxvf node-v0.10.24.tar.gz
```

3、 编译安装

```
cd node-v0.10.24
./configure --prefix=/usr/local/node/0.10.24
make
make install
```

4、 配置NODE_HOME，进入profile编辑环境变量

```
vim /etc/profile
```

设置nodejs环境变量，在*export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL* 一行的上面添加如下内容:

```
#set for nodejs
export NODE_HOME=/usr/local/node/0.10.24
export PATH=$NODE_HOME/bin:$PATH
```

:wq保存并退出，编译/etc/profile 使配置生效

```
source /etc/profile
```

验证是否安装配置成功

```
node -v
```

输出 v0.10.24 表示配置成功

npm模块安装路径

```
/usr/local/node/0.10.24/lib/node_modules/
```

注：Nodejs 官网提供了编译好的Linux二进制包，你也可以下载下来直接应用。

Node.js 创建第一个应用

如果我们使用PHP来编写后端的代码时，需要Apache 或者 Nginx 的HTTP 服务器，并配上 mod_php5 模块和php-cgi。

从这个角度看，整个"接收 HTTP 请求并提供 Web 页面"的需求根本不需 要 PHP 来处理。

不过对 Node.js 来说，概念完全不一样了。使用 Node.js 时，我们不仅仅 在实现一个应用，同时还实现了整个 HTTP 服务器。事实上，我们的 Web 应用以及对应的 Web 服务器基本上是一样的。

在我们创建 Node.js 第一个 "Hello, World!" 应用前，让我们先了解下 Node.js 应用是由哪几部分组成的：

1. 引入 **required** 模块：我们可以使用 **require** 指令来载入 Node.js 模块。
2. 创建服务器：服务器可以监听客户端的请求，类似于 Apache 、Nginx 等 HTTP 服务器。
3. 接收请求与响应请求 服务器很容易创建，客户端可以使用浏览器或终端发送 HTTP 请求，服务器接收请求后返回响应数据。

创建 Node.js 应用

步骤一、引入 required 模块

我们使用 **require** 指令来载入 http 模块，并将实例化的 HTTP 赋值给变量 http，实例如下:

```
var http =  require("http");
```

步骤一、创建服务器

接下来我们使用 http.createServer() 方法创建服务器，并使用 listen 方法绑定 8888 端口。 函数通过 request, response 参数来接收和响应数据。

实例如下，在你项目的根目录下创建一个叫 server.js 的文件，并写入以下代码：

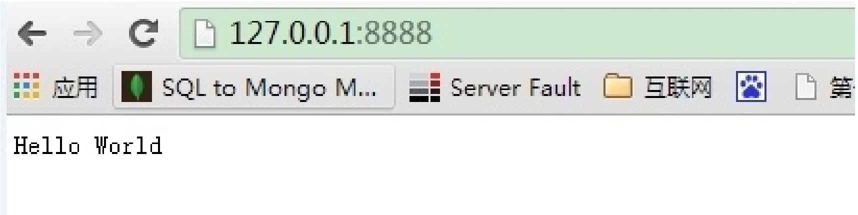
```
var http =  require('http'); http.createServer(function  (request, response)  {  // 发送 H
```

以上代码我们完成了一个可以工作的 HTTP 服务器。

使用 **node** 命令执行以上的代码：

```
node server.js Server running at http://127.0.0.1:8888/
```

接下来，打开浏览器访问 http://127.0.0.1:8888/，你会看到一个写着 "Hello World"的网页。

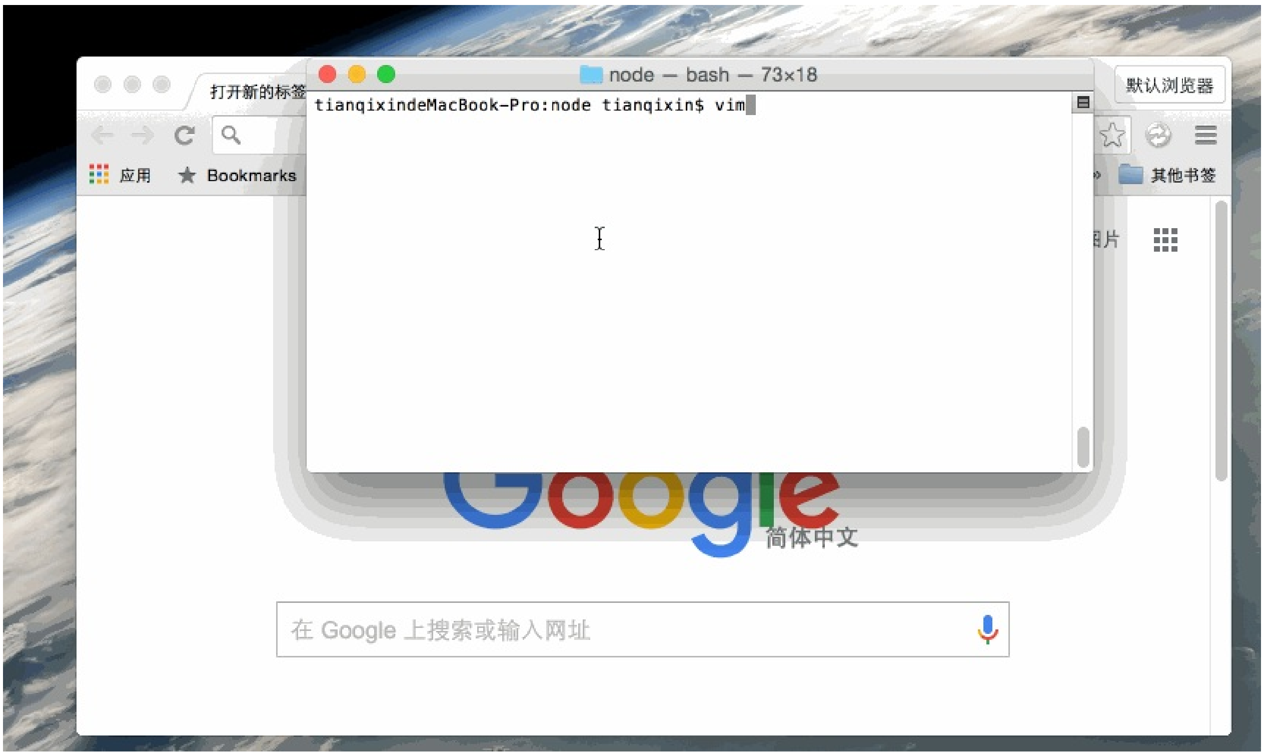


分析**Node.js** 的 **HTTP** 服务器：

- 第一行请求（require）Node.js 自带的 http 模块，并且把它赋值给 http 变量。
- 接下来我们调用 http 模块提供的函数： createServer 。这个函数会返回 一个对象，这个对象有一个叫做 listen 的方法，这个方法有一个数值参数， 指定这个 HTTP 服务器监听的端口号。

Gif 实例演示

接下来我们通过 Gif 图为大家演示实例操作：



NPM 使用介绍

NPM是随同NodeJS一起安装的包管理工具，能解决NodeJS代码部署上的很多问题，常见的使用场景有以下几种：

- 允许用户从NPM服务器下载别人编写的第三方包到本地使用。
- 允许用户从NPM服务器下载并安装别人编写的命令行程序到本地使用。
- 允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

由于新版的nodejs已经集成了npm，所以之前npm也一并安装好了。同样可以通过输入 **"npm -v"** 来测试是否成功安装。命令如下，出现版本提示表示安装成功:

```
$ npm -v
2.3.0
```

如果你安装的是旧版本的 npm，可以很容易得通过 npm 命令来升级，命令如下：

```
$ sudo npm install npm -g
/usr/local/bin/npm -> /usr/local/lib/node_modules/npm/bin/npm-cli.js
npm@2.14.2 /usr/local/lib/node_modules/npm
```

使用 npm 命令安装模块

npm 安装 Node.js 模块语法格式如下：

```
$ npm install <Module Name>
```

以下实例，我们使用 npm 命令安装常用的 Node.js web框架模块 **express**:

```
$ npm install express
```

安装好之后，express 包就放在了工程目录下的 node_modules 目录中，因此在代码中只需要通过 **require('express')** 的方式就好，无需指定第三方包路径。

```
var express = require('express');
```

全局安装与本地安装

npm 的包安装分为本地安装（local）、全局安装（global）两种，从敲的命令行来看，差别只是有没有-g而已，比如

```
npm install express          # 本地安装
npm install express -g   # 全局安装
```

如果出现以下错误：

```
npm err! Error: connect ECONNREFUSED 127.0.0.1:8087
```

解决办法为：

```
$ npm config set proxy null
```

本地安装

- 1. 将安装包放在 ./node_modules 下（运行 npm 命令时所在的目录），如果没有 node_modules 目录，会在当前执行 npm 命令的目录下生成 node_modules 目录。
- 2. 可以通过 require() 来引入本地安装的包。

全局安装

- 1. 将安装包放在 /usr/local 下。
- 2. 可以直接在命令行里使用。
- 3. 不能通过 require() 来引入本地安装的包。

接下来我们使用全局方式安装 express

```
$ npm install express -g
```

安装过程输出如下内容，第一行输出了模块的版本号及安装位置。

```
express@4.13.3 node_modules/express
├── escape-html@1.0.2
├── range-parser@1.0.2
├── merge-descriptors@1.0.0
├── array-flatten@1.1.1
├── cookie@0.1.3
├── utils-merge@1.0.0
├── parseurl@1.3.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.3.0
├── vary@1.0.1
├── path-to-regexp@0.1.7
├── content-type@1.0.1
├── etag@1.7.0
├── serve-static@1.10.0
├── content-disposition@0.5.0
├── depd@1.0.1
├── qs@4.0.0
├── finalhandler@0.4.0 (unpipe@1.0.0)
├── on-finished@2.3.0 (ee-first@1.1.1)
├── proxy-addr@1.0.8 (forwarded@0.1.0, ipaddr.js@1.0.1)
├── debug@2.2.0 (ms@0.7.1)
├── type-is@1.6.8 (media-typer@0.3.0, mime-types@2.1.6)
├── accepts@1.2.12 (negotiator@0.5.3, mime-types@2.1.6)
└── send@0.13.0 (destroy@1.0.3, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-errors@1.3.1)
```

你可以使用以下命令来查看所有全局安装的模块：

```
$ npm ls -g
```

使用 package.json

package.json 位于模块的目录下，用于定义包的属性。接下来让我们来看下 express 包的 package.json 文件，位于 node_modules/express/package.json 内容：

```
{
  "name": "express",
  "description": "Fast, unopinionated, minimalist web framework",
  "version": "4.13.3",
  "author": {
    "name": "TJ Holowaychuk",
    "email": "tj@vision-media.ca"
  },
  "contributors": [
    {
      "name": "Aaron Heckmann",
      "email": "aaron.heckmann+github@gmail.com"
    },
    {
      "name": "Ciaran Jessup",
      "email": "ciaranj@gmail.com"
    },
    {
      "name": "Douglas Christopher Wilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "Guillermo Rauch",
      "email": "rauchg@gmail.com"
```

```
    },
    {
      "name": "Jonathan Ong",
      "email": "me@jongleberry.com"
    },
    {
      "name": "Roman Shtylman",
      "email": "shtylman+expressjs@gmail.com"
    },
    {
      "name": "Young Jae Sim",
      "email": "hanul@hanul.me"
    }
  ],
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "git+https://github.com/strongloop/express.git"
  },
  "homepage": "http://expressjs.com/",
  "keywords": [
    "express",
    "framework",
    "sinatra",
    "web",
    "rest",
    "restful",
    "router",
    "app",
    "api"
  ],
  "dependencies": {
    "accepts": "~1.2.12",
    "array-flatten": "1.1.1",
    "content-disposition": "0.5.0",
    "content-type": "~1.0.1",
    "cookie": "0.1.3",
    "cookie-signature": "1.0.6",
    "debug": "~2.2.0",
    "depd": "~1.0.1",
    "escape-html": "1.0.2",
    "etag": "~1.7.0",
    "finalhandler": "0.4.0",
    "fresh": "0.3.0",
    "merge-descriptors": "1.0.0",
    "methods": "~1.1.1",
    "on-finished": "~2.3.0",
    "parseurl": "~1.3.0",
    "path-to-regexp": "0.1.7",
    "proxy-addr": "~1.0.8",
    "qs": "4.0.0",
    "range-parser": "~1.0.2",
    "send": "0.13.0",
    "serve-static": "~1.10.0",
    "type-is": "~1.6.6",
    "utils-merge": "1.0.0",
    "vary": "~1.0.1"
  },
  "devDependencies": {
    "after": "0.8.1",
    "ejs": "2.3.3",
    "istanbul": "0.3.17",
    "marked": "0.3.5",
    "mocha": "2.2.5",
    "should": "7.0.2",
    "supertest": "1.0.1",
    "body-parser": "~1.13.3",
    "connect-redis": "~2.4.1",
    "cookie-parser": "~1.3.5",
    "cookie-session": "~1.2.0",
    "express-session": "~1.11.3",
    "jade": "~1.11.0",
```

```
      "method-override": "~2.3.5",
      "morgan": "~1.6.1",
      "multiparty": "~4.1.2",
      "vhost": "~3.0.1"
    },
    "engines": {
      "node": ">= 0.10.0"
    },
    "files": [
      "LICENSE",
      "History.md",
      "Readme.md",
      "index.js",
      "lib/"
    ],
    "scripts": {
      "test": "mocha --require test/support/env --reporter spec --bail --check-leaks test/
      "test-ci": "istanbul cover node_modules/mocha/bin/_mocha --report lcovonly -- --requi
      "test-cov": "istanbul cover node_modules/mocha/bin/_mocha -- --require test/support/e
      "test-tap": "mocha --require test/support/env --reporter tap --check-leaks test/ test
    },
    "gitHead": "ef7ad681b245fba023843ce94f6bcb8e275bbb8e",
    "bugs": {
      "url": "https://github.com/strongloop/express/issues"
    },
    "_id": "express@4.13.3",
    "_shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
    "_from": "express@*",
    "_npmVersion": "1.4.28",
    "_npmUser": {
      "name": "dougwilson",
      "email": "doug@somethingdoug.com"
    },
    "maintainers": [
      {
        "name": "tjholowaychuk",
        "email": "tj@vision-media.ca"
      },
      {
        "name": "jongleberry",
        "email": "jonathanrichardong@gmail.com"
      },
      {
        "name": "dougwilson",
        "email": "doug@somethingdoug.com"
      },
      {
        "name": "rfeng",
        "email": "enjoyjava@gmail.com"
      },
      {
        "name": "aredridel",
        "email": "aredridel@dinhe.net"
      },
      {
        "name": "strongloop",
        "email": "callback@strongloop.com"
      },
      {
        "name": "defunctzombie",
        "email": "shtylman@gmail.com"
      }
    ],
    "dist": {
      "shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
      "tarball": "http://registry.npmjs.org/express/-/express-4.13.3.tgz"
    },
    "directories": {},
    "_resolved": "https://registry.npmjs.org/express/-/express-4.13.3.tgz",
    "readme": "ERROR: No README data found!"
  }
```

Package.json 属性说明

- **name** - 包名。
- **version** - 包的版本号。
- **description** - 包的描述。
- **homepage** - 包的官网 url 。
- **author** - 包的作者姓名。
- **contributors** - 包的其他贡献者姓名。
- **dependencies** - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node_module 目录下。
- **repository** - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。
- **main** - main 字段是一个模块ID，它是一个指向你程序的主要项目。就是说，如果你包的名字叫 express，然后用户安装它，然后require("express")。
- **keywords** - 关键字

卸载模块

我们可以使用以下命令来卸载 Node.js 模块。

```
$ npm uninstall express
```

卸载后，你可以到 /node_modules/ 目录下查看包是否还存在，或者使用以下命令查看：

```
$ npm ls
```

更新模块

我们可以使用以下命令更新模块：

```
$ npm update express
```

搜索模块

使用以下来搜索模块：

```
$ npm search express
```

创建模块

创建模块，package.json 文件是必不可少的。我们可以使用 NPM 生成 package.json 文件，生成的文件包含了基本的结果。

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (node_modules) runoob                   # 模块名
version: (1.0.0)
description: Node.js 测试模块(www.runoob.com)  # 描述
entry point: (index.js)
test command: make test
git repository: https://github.com/runoob/runoob.git  # Github 地址
keywords:
author:
license: (ISC)
About to write to ……/node_modules/package.json:      # 生成地址

{
  "name": "runoob",
  "version": "1.0.0",
  "description": "Node.js 测试模块(www.runoob.com)",
  ……
}

Is this ok? (yes) yes
```

以上的信息，你需要根据你自己的情况输入。在最后输入 "yes" 后会生成 package.json 文件。

接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

```
$ npm adduser
Username: mcmohd
Password:
Email: (this IS public) mcmohd@gmail.com
```

接下来我们就用以下命令来发布模块：

```
$ npm publish
```

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

版本号

使用NPM下载和发布代码时都会接触到版本号。NPM使用语义版本号来管理代码，这里简单介绍一下。

语义版本号分为X.Y.Z三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新。

- 如果只是修复bug，需要更新Z位。
- 如果是新增了功能，但是向下兼容，需要更新Y位。
- 如果有大变动，向下不兼容，需要更新X位。

版本号有了这个保证后，在申明第三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如"argv": "0.0.x"表示依赖于0.0.x系列的最新版argv。

NPM支持的所有版本号范围指定方式可以查看官方文档。

NPM 常用命令

除了本章介绍的部分外，NPM还提供了很多功能，package.json里也有很多其它有用的字段。

除了可以在npmjs.org/doc/查看官方文档外，这里再介绍一些NPM常用命令。

NPM提供了很多命令，例如install和publish，使用npm help可查看所有命令。

- NPM提供了很多命令，例如 `install` 和 `publish` ，使用 `npm help` 可查看所有命令。
- 使用 `npm help <command>` 可查看某条命令的详细帮助，例如 `npm help install` 。
- 在 `package.json` 所在目录下使用 `npm install . -g` 可先在本地安装当前命令行程序，可用于发布前的本地测试。
- 使用 `npm update <package>` 可以把当前目录下 `node_modules` 子目录里边的对应模块更新至最新版本。
- 使用 `npm update <package> -g` 可以把全局安装的对应命令行程序更新至最新版。
- 使用 `npm cache clear` 可以清空NPM本地缓存，用于对付使用相同版本号发布新版本代码的人。
- 使用 `npm unpublish <package>@<version>` 可以撤销发布自己发布过的某个版本代码。

Node.js REPL(交互式解释器)

Node.js REPL(Read Eval Print Loop:交互式解释器) 表示一个电脑的环境，类似 Window 系统的终端或 Unix/Linux shell，我们可以在终端中输入命令，并接收系统的响应。

Node 自带了交互式解释器，可以执行以下任务：

- 读取 - 读取用户输入，解析输入了Javascript 数据结构并存储在内存中。
- 执行 - 执行输入的数据结构
- 打印 - 输出结果
- 循环 - 循环操作以上步骤直到用户两次按下 **ctrl-c** 按钮退出。

Node 的交互式解释器可以很好的调试 Javascript 代码。

开始学习 REPL

我们可以输入以下命令来启动 Node 的终端：

```
$ node
>
```

这时我们就可以在 > 后输入简单的表达式，并按下回车键来计算结果。

简单的表达式运算

接下来让我们在 Node.js REPL 的命令行窗口中执行简单的数学运算：

```
$ node
> 1 +4
5
> 5 / 2
2.5
> 3 * 6
18
> 4 - 1
3
> 1 + ( 2 * 3 ) - 4
3
>
```

使用变量

你可以将数据存储在变量中，并在你需要的使用它。

变量声明需要使用 **var** 关键字，如果没有使用 var 关键字变量会直接打印出来。

使用 **var** 关键字的变量可以使用 console.log() 来输出变量。

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello World
undefined
> console.log("www.runoob.com")
www.runoob.com
undefined
```

多行表达式

Node REPL 支持输入多行表达式，这就有点类似 JavaScript。接下来让我们来执行一个 do-while 循环：

```
$ node
> var x = 0
undefined
> do {
... x++;
... console.log("x: " + x);
... } while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

... 三个点的符号是系统自动生成的，你回车换行后即可。Node 会自动检测是否为连续的表达式。

下划线(_)变量

你可以使用下划线(_)获取表达式的运算结果：

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

REPL 命令

- **ctrl + c** - 退出当前终端。
- **ctrl + c 按下两次** - 退出 Node REPL。
- **ctrl + d** - 退出 Node REPL.
- **向上/向下 键** - 查看输入的历史命令
- **tab 键** - 列出当前命令
- **.help** - 列出使用命令
- **.break** - 退出多行表达式
- **.clear** - 退出多行表达式
- **.save *filename*** - 保存当前的 Node REPL 会话到指定文件
- **.load *filename*** - 载入当前 Node REPL 会话的文件内容。

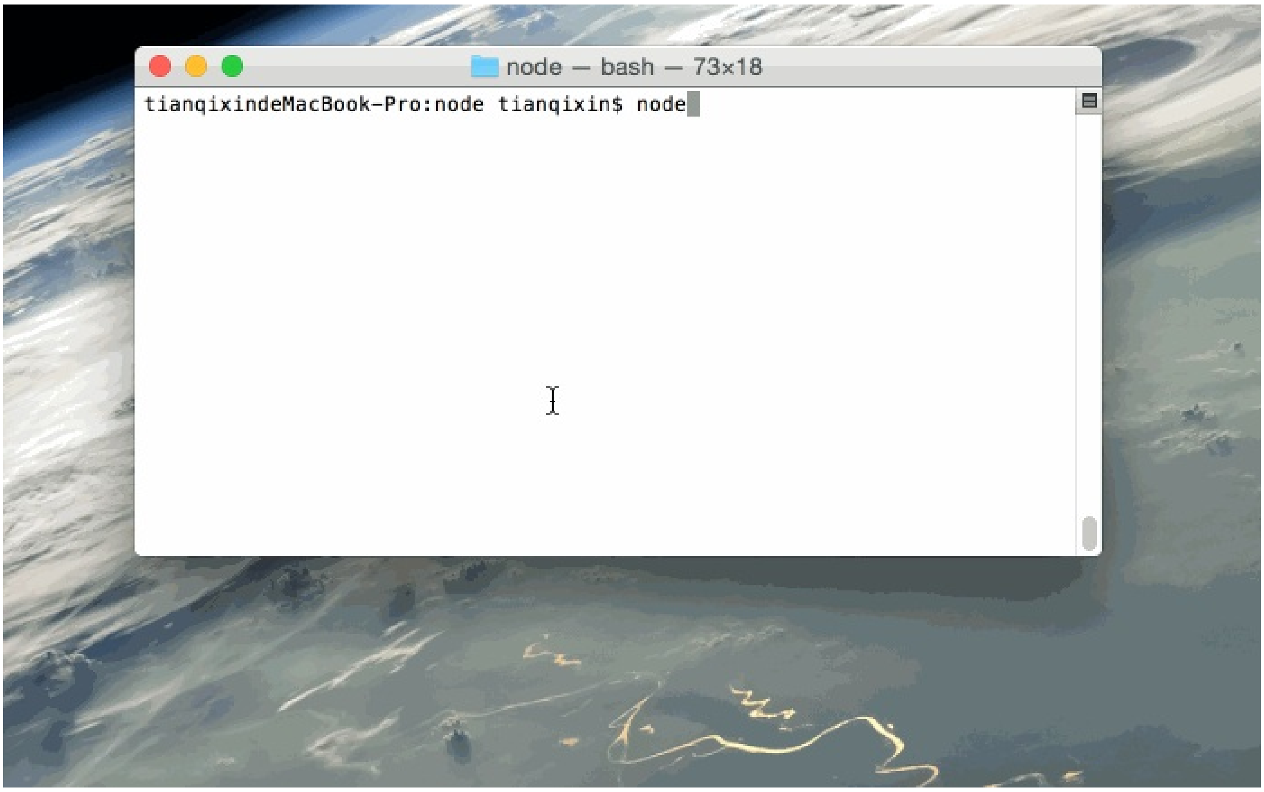
停止 REPL

前面我们已经提到按下两次 **ctrl + c** 建就能退出 REPL:

```
$ node
>
(^C again to quit)
>
```

Gif 实例演示

接下来我们通过 Gif 图为大家演示实例操作：


node — bash — 73×18
tianqixindeMacBook-Pro:node tianqixin$ node

Node.js 回调函数

Node.js 异步编程的直接体现就是回调。

异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。

回调函数在完成任务后就会被调用，Node 使用了大量的回调函数，Node 所有 API 都支持回调函数。

例如，我们可以一边读取文件，一边执行其他命令，在文件读取完成后，我们将文件内容作为回调函数的参数返回。这样在执行代码时就没有阻塞或等待文件 I/O 操作。这就大大提高了 Node.js 的性能，可以处理大量的并发请求。

阻塞代码实例

创建一个文件 input.txt ，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("程序执行结束!");
```

以上代码执行结果如下：

```
$ node main.js
菜鸟教程官网地址：www.runoob.com

程序执行结束!
```

非阻塞代码实例

创建一个文件 input.txt ，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
});

console.log("程序执行结束!");
```

以上代码执行结果如下：

```
$ node main.js
程序执行结束!
菜鸟教程官网地址：www.runoob.com
```

以上两个实例我们了解了阻塞与非阻塞调用的不同。第一个实例在文件读取完后才执行完程序。 第二个实例我们呢不需要等待文件读取完，这样就可以在读取文件时同时执行接下来的代码，大大提高了程序的性能。

因此，阻塞按是按顺序执行的，而非阻塞是不需要按顺序的，所以如果需要处理回调函数的参数，我们就需要写在回调函数内。

Node.js 事件循环

Node.js 是单进程单线程应用程序，但是通过事件和回调支持并发，所以性能非常高。

Node.js 的每一个 API 都是异步的，并作为一个独立线程运行，使用异步函数调用，并处理并发。

Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。

Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数.

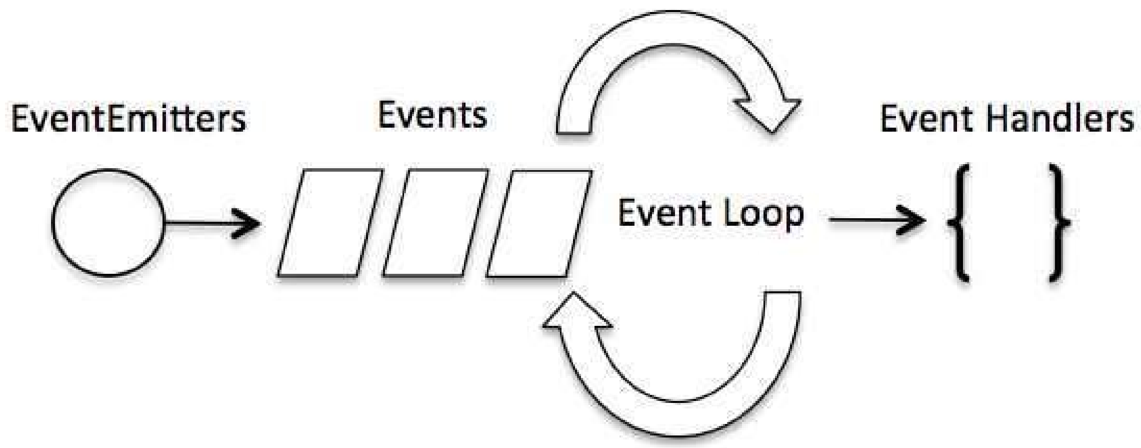
事件驱动程序

Node.js 使用事件驱动模型，当web server接收到请求，就把它关闭然后进行处理，然后去服务下一个web请求。

当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。

这个模型非常高效可扩展性非常强，因为webserver一直接受请求而不等待任何读写操作。（这也被称之为非阻塞式IO或者事件驱动IO）

在事件驱动模型中，会生成一个主循环来监听事件，当检测到事件时触发回调函数。



整个事件驱动的流程就是这么实现的，非常简洁。有点类似于观察者模式，事件相当于一个主题(Subject)，而所有注册到这个事件上的处理函数相当于观察者(Observer)。

Node.js 有多个内置的事件，我们可以通过引入 events 模块，并通过实例化 EventEmitter 类来绑定和监听事件，如下实例：

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();
```

以下程序绑定事件处理程序：

```
// 绑定事件及事件的处理程序
eventEmitter.on('eventName', eventHandler);
```

我们可以通过程序触发事件：

```
// 触发事件
eventEmitter.emit('eventName');
```

实例

创建 main.js 文件，代码如下所示：

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();

// 创建事件处理程序
var connectHandler = function connected() {
   console.log('连接成功。');

   // 触发 data_received 事件
   eventEmitter.emit('data_received');
}

// 绑定 connection 事件处理程序
eventEmitter.on('connection', connectHandler);

// 使用匿名函数绑定 data_received 事件
eventEmitter.on('data_received', function(){
   console.log('数据接收成功。');
});

// 触发 connection 事件
eventEmitter.emit('connection');

console.log("程序执行完毕。");
```

接下来让我们执行以上代码：

```
$ node main.js
连接成功。
数据接收成功。
程序执行完毕。
```

Node 应用程序是如何工作的？

在 Node 应用程序中，执行异步操作的函数将回调函数作为最后一个参数， 回调函数接收错误对象作为第一个参数。

接下来让我们来重新看下前面的实例，创建一个 input.txt ,文件内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件，代码如下：

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
   if (err){
      console.log(err.stack);
      return;
   }
   console.log(data.toString());
});
console.log("程序执行完毕");
```

以上程序中 fs.readFile() 是异步函数用于读取文件。 如果在读取文件过程中发生错误，错误 err 对象就会输出错误信息。

如果没发生错误，readFile 跳过 err 对象的输出，文件内容就通过回调函数输出。

执行以上代码，执行结果如下：

```
程序执行完毕
菜鸟教程官网地址：www.runoob.com
```

接下来我们删除 input.txt 文件，执行结果如下所示：

```
程序执行完毕
Error: ENOENT, open 'input.txt'
```

因为文件 input.txt 不存在，所以输出了错误信息。

Node.js EventEmitter

Node.js 所有的异步 I/O 操作在完成时都会发送一个事件到事件队列。

Node.js里面的许多对象都会分发事件：一个net.Server对象会在每次有新连接时分发一个事件， 一个fs.readStream对象会在文件被打开的时候发出一个事件。 所有这些产生事件的对象都是 events.EventEmitter 的实例。

EventEmitter 类

events 模块只提供了一个对象： events.EventEmitter。EventEmitter 的核心就是事件触发与事件监听器功能的封装。

你可以通过require("events");来访问该模块。

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();
```

EventEmitter 对象如果在实例化时发生错误，会触发 'error' 事件。当添加新的监听器时，'newListener' 事件会触发，当监听器被移除时，'removeListener' 事件被触发。

下面我们用一个简单的例子说明 EventEmitter 的用法：

```
//event.js 文件
var EventEmitter = require('events').EventEmitter;
var event = new EventEmitter();
event.on('some_event', function() {
    console.log('some_event 事件触发');
});
setTimeout(function() {
    event.emit('some_event');
}, 1000);
```

执行结果如下：

运行这段代码，1 秒后控制台输出了 **'some_event 事件触发'**。其原理是 event 对象注册了事件 some_event 的一个监听器，然后我们通过 setTimeout 在 1000 毫秒以后向 event 对象发送事件 some_event，此时会调用some_event 的监听器。

```
$ node event.js
some_event 事件触发
```

EventEmitter 的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持 若干个事件监听器。

当事件触发时，注册到这个事件的事件监听器被依次调用，事件参数作为回调函数参数传递。

让我们以下面的例子解释这个过程：

```
//event.js 文件
var events = require('events');
var emitter = new events.EventEmitter();
emitter.on('someEvent', function(arg1, arg2) {
    console.log('listener1', arg1, arg2);
});
emitter.on('someEvent', function(arg1, arg2) {
    console.log('listener2', arg1, arg2);
});
emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');
```

执行以上代码，运行的结果如下：/p>

```
$ node event.js
listener1 arg1 参数 arg2 参数
listener2 arg1 参数 arg2 参数
```

以上例子中，emitter 为事件 someEvent 注册了两个事件监听器，然后触发了 someEvent 事件。

运行结果中可以看到两个事件监听器回调函数被先后调用。 这就是EventEmitter最简单的用法。

EventEmitter 提供了多个属性，如 **on** 和 **emit**。**on** 函数用于绑定事件函数，**emit** 属性用于触发一个事件。接下来我们来具体看下 EventEmitter 的属性介绍。

方法

方法	
addListener(event, listener)	为指定事件添加一个监听器到监听器数组的尾部。
on(event, listener)	为指定事件注册一个监听器，接受一个字符串 event 和一
once(event, listener)	为指定事件注册一个单次监听器，即 监听器最多只会触发 `server.once('connection', function (stream) { console.l`
removeListener(event, listener)	移除指定事件的某个监听器，监听器 必须是该事件已经注 `var callback = function(stream) { console.log('someone`
removeAllListeners([event])	移除所有事件的所有监听器， 如果指定事件，则移除指定
setMaxListeners(n)	默认情况下， EventEmitters 如果你添加的监听器超过 10
listeners(event)	返回指定事件的监听器数组。
emit(event, [arg1], [arg2], [...])	按参数的顺序执行每个监听器，如果事件有注册监听返回

类方法

方法	描述
listenerCount(emitter, event)	返回指定事件的监听器数量。

事件

事件	描述
newListener	**event** - 字符串，事件名称 **listener** - 处理事件函数 该事件在添加新监听器时被触发。
removeListener	**event** - 字符串，事件名称 **listener** - 处理事件函数 从指定监听器数组中删除一个监听器。需要注意的是，此操作将会改变处于被删监听器之后的那些监听器的索引。

实例

以下实例通过 connection（连接）事件演示了 EventEmitter 类的应用。

创建 main.js 文件，代码如下：

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// 监听器 #1
var listner1 = function listner1() {
   console.log('监听器 listner1 执行。');
}

// 监听器 #2
var listner2 = function listner2() {
  console.log('监听器 listner2 执行。');
}

// 绑定 connection 事件，处理函数为 listner1
eventEmitter.addListener('connection', listner1);

// 绑定 connection 事件，处理函数为 listner2
eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connectio
console.log(eventListeners + " 监听器监听连接事件。");

// 处理 connection 事件
eventEmitter.emit('connection');

// 移除监绑定的 listner1 函数
eventEmitter.removeListener('connection', listner1);
console.log("listner1 不再受监听。");

// 触发连接事件
eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " 监听器监听连接事件。");

console.log("程序执行完毕。");
```

以上代码，执行结果如下所示：

```
$ node main.js
2 监听器监听连接事件。
监听器 listner1 执行。
监听器 listner2 执行。
listner1 不再受监听。
监听器 listner2 执行。
1 监听器监听连接事件。
程序执行完毕。
```

error 事件

EventEmitter 定义了一个特殊的事件 error，它包含了错误的语义，我们在遇到 异常的时候通常会触发 error 事件。

当 error 被触发时，EventEmitter 规定如果没有响 应的监听器，Node.js 会把它当作异常，退出程序并输出错误信息。

我们一般要为会触发 error 事件的对象设置监听器，避免遇到错误后整个程序崩溃。例如：

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.emit('error');
```

运行时会显示以下错误：

```
node.js:201
throw e; // process.nextTick error, or 'error' event on first tick
^
Error: Uncaught, unspecified 'error' event.
at EventEmitter.emit (events.js:50:15)
at Object.<anonymous> (/home/byvoid/error.js:5:9)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

继承 EventEmitter

大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、 http 在内的，只要是支持事件响应的核心模块都是 EventEmitter 的子类。

为什么要这样做呢？原因有两点：

首先，具有某个实体功能的对象实现事件符合语义， 事件的监听和发射应该是一个对象的方法。

其次 JavaScript 的对象机制是基于原型的，支持 部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

Node.js Buffer(缓冲区)

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

但在处理像TCP流或文件流时，必须使用到二进制数据。因此在 Node.js中，定义了一个 Buffer 类，该类用来创建一个专门存放二进制数据的缓存区。

在 Node.js 中，Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理I/O操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

创建 Buffer 类

Node Buffer 类可以通过多种方式来创建。

方法 1

创建长度为 10 字节的 Buffer 实例：

```
var buf = new Buffer(10);
```

方法 2

通过给定的数组创建 Buffer 实例：

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

方法 3

通过一个字符串来创建 Buffer 实例：

```
var buf = new Buffer("www.runoob.com", "utf-8");
```

utf-8 是默认的编码方式，此外它同样支持以下编码："ascii", "utf8", "utf16le", "ucs2", "base64" 和 "hex"。

写入缓冲区

语法

写入 Node 缓冲区的语法如下所示：

```
buf.write(string[, offset][, length][, encoding])
```

参数

参数描述如下：

- **string** - 写入缓冲区的字符串。
- **offset** - 缓冲区开始写入的索引值，默认为 0 。
- **length** - 写入的字节数，默认为 buffer.length
- **encoding** - 使用的编码。默认为 'utf8' 。

返回值

返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。

实例

```
buf = new Buffer(256);
len = buf.write("www.runoob.com");

console.log("写入字节数 : "+  len);
```

执行以上代码，输出结果为：

```
$node main.js
写入字节数 : 14
```

从缓冲区读取数据

语法

读取 Node 缓冲区数据的语法如下所示：

```
buf.toString([encoding][, start][, end])
```

参数

参数描述如下：

- **encoding** - 使用的编码。默认为 'utf8' 。
- **start** - 指定开始读取的索引位置，默认为 0。
- **end** - 结束位置，默认为缓冲区的末尾。

返回值

解码缓冲区数据并使用指定的编码返回字符串。

实例

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // 输出: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // 输出: abcde
console.log( buf.toString('utf8',0,5));    // 输出: abcde
console.log( buf.toString(undefined,0,5)); // 使用 'utf8' 编码, 并输出: abcde
```

执行以上代码，输出结果为：

```
$ node main.js
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

将 Buffer 转换为 JSON 对象

语法

将 Node Buffer 转换为 JSON 对象的函数语法格式如下：

```
buf.toJSON()
```

返回值

返回 JSON 对象。

实例

```
var buf = new Buffer('www.runoob.com');
var json = buf.toJSON(buf);

console.log(json);
```

执行以上代码，输出结果为：

```
[ 119, 119, 119, 46, 114, 117, 110, 111, 111, 98, 46, 99, 111, 109 ]
```

缓冲区合并

语法

Node 缓冲区合并的语法如下所示：

```
Buffer.concat(list[, totalLength])
```

参数

参数描述如下：

- **list** - 用于合并的 Buffer 对象数组列表。
- **totalLength** - 指定合并后Buffer对象的总长度。

返回值

返回一个多个成员合并的新 Buffer 对象。

实例

```
var buffer1 = new Buffer('菜鸟教程 ');
var buffer2 = new Buffer('www.runoob.com');
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 内容: " + buffer3.toString());
```

执行以上代码，输出结果为：

```
buffer3 内容: 菜鸟教程 www.runoob.com
```

缓冲区比较

语法

Node Buffer 比较的函数语法如下所示：

```
buf.compare(otherBuffer);
```

参数

参数描述如下：

- **otherBuffer** - 与 **buf** 对象比较的另外一个 Buffer 对象。

返回值

返回一个数字，表示 **buf** 在 **otherBuffer** 之前，之后或相同。

实例

```
var buffer1 = new Buffer('ABC');
var buffer2 = new Buffer('ABCD');
var result = buffer1.compare(buffer2);

if(result < 0) {
   console.log(buffer1 + " 在 " + buffer2 + "之前");
}else if(result == 0){
   console.log(buffer1 + " 与 " + buffer2 + "相同");
}else {
   console.log(buffer1 + " 在 " + buffer2 + "之后");
}
```

执行以上代码，输出结果为：

```
ABC在ABCD之前
```

拷贝缓冲区

语法

Node 缓冲区拷贝语法如下所示：

```
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
```

参数

参数描述如下：

- **targetBuffer** - 要拷贝的 Buffer 对象。
- **targetStart** - 数字, 可选, 默认: 0
- **sourceStart** - 数字, 可选, 默认: 0
- **sourceEnd** - 数字, 可选, 默认: buffer.length

返回值

没有返回值。

实例

```
var buffer1 = new Buffer('ABC');
// 拷贝一个缓冲区
var buffer2 = new Buffer(3);
buffer1.copy(buffer2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ABC
```

缓冲区裁剪

Node 缓冲区裁剪语法如下所示：

```
buf.slice([start][, end])
```

参数

参数描述如下：

- **start** - 数字, 可选, 默认: 0
- **end** - 数字, 可选, 默认: buffer.length

返回值

返回一个新的缓冲区，它和旧缓冲区指向同一块内存，但是从索引 start 到 end 的位置剪切。

实例

```
var buffer1 = new Buffer('runoob');
// 剪切缓冲区
var buffer2 = buffer1.slice(0,2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ru
```

缓冲区长度

语法

Node 缓冲区长度计算语法如下所示：

```
buf.length;
```

返回值

返回 Buffer 对象所占据的内存长度。

实例

```
var buffer = new Buffer('www.runoob.com');
//  缓冲区长度
console.log("buffer length: " + buffer.length);
```

执行以上代码，输出结果为：

```
buffer length: 14
```

方法参考手册

以下列出了 Node.js Buffer 模块常用的方法（注意有些方法在旧版本是没有的）：

方法	描述
new Buffer(size)	分配一个新的 size 大小单位为8位字节的 buffer。 注意，size 将会抛出异常 RangeError。
new Buffer(buffer)	拷贝参数 buffer 的数据到 Buffer 实例。

new Buffer(str[, encoding])	分配一个新的 buffer ，其中包含着传入的 str 字符串。 enco
buf.length	返回这个 buffer 的 bytes 数。注意这未必是 buffer 里面内容 分配的内存数，它不会随着这个 buffer 对象内容的改变而改
buf.write(string[, offset][, length][, encoding])	根据参数 offset 偏移量和指定的 encoding 编码方式，将参数 偏移量默认值是 0, encoding 编码方式默认是 utf8。 length 大小。 返回 number 类型，表示写入了多少 8 位字节流。如 个 string，它将只会只写入部分字符串。 length 默认是 buff 出现写入部分字符。
buf.writeUIntLE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， `var b = new Buffer(6); b.writeUIntBE(0x1234567890ab, 0, 6` noAssert 值为 true 时，不再验证 value 和 offset 的有效性。
buf.writeUIntBE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.writeIntLE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.writeIntBE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.readUIntLE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readUIntBE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readIntLE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readIntBE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.toString([encoding][, start][, end])	根据 encoding 参数（默认是 'utf8'）返回一个解码过的 strin start (默认是 0) 和 end (默认是 buffer.length)作为取值范围。
buf.toJSON()	将 Buffer 实例转换为 JSON 对象。
buf[index]	获取或设置指定的字节。返回值代表一个字节，所以返回值 0xFF 或者十进制0至 255。
buf.equals(otherBuffer)	比较两个缓冲区是否相等，如果是返回 true，否则返回 fals
buf.compare(otherBuffer)	比较两个 Buffer 对象，返回一个数字，表示 buf 在 otherBu
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])	buffer 拷贝，源和目标可以相同。 targetStart 目标开始偏移 是 0。 sourceEnd 源结束位置偏移默认是源的长度 buffer.le
buf.slice([start][, end])	剪切 Buffer 对象，根据 start(默认是 0) 和 end (默认是 buff 负的索引是从 buffer 尾部开始计算的。

buf.readUInt8(offset[, noAssert])	根据指定的偏移量，读取一个有符号 8 位整数。若参数 noA 移量参数。 如果这样 offset 可能会超出buffer 的末尾。默认
buf.readUInt16LE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt16BE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt32LE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt32BE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readInt8(offset[, noAssert])	根据指定的偏移量，读取一个 signed 8 位整数。 若参数 no 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默
buf.readInt16LE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 格式读取一个 signe 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readInt16BE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 格式读取一个 signe 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readInt32LE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readInt32BE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readFloatLE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readFloatBE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readDoubleLE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian字节序格式读取一个 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readDoubleBE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian字节序格式读取一个 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.writeUInt8(value, offset[, noAssert])	根据传入的 offset 偏移量将 value 写入 buffer。注意：value 数。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数 者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。

	否则不要使用。默认是 false。
buf.writeUInt16LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的有符号 16 位整数。 若参数 noAssert 为 true 将不会数。 这意味着 value 可能过大，或者 offset 可能会超出buff除非你对这个参数非常有把握，否则尽量不要使用。默认是
buf.writeUInt16BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的有符号 16 位整数。 若参数 noAssert 为 true 将不会数。 这意味着 value 可能过大，或者 offset 可能会超出buff除非你对这个参数非常有把握，否则尽量不要使用。默认是
buf.writeUInt32LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会这意味着value 可能过大，或者offset可能会超出buffer的末对这个参数非常有把握，否则尽量不要使用。默认是 false。
buf.writeUInt32BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会这意味着value 可能过大，或者offset可能会超出buffer的末对这个参数非常有把握，否则尽量不要使用。默认是 false。
buf.writeInt8(value, offset[, noAssert])	
buf.writeInt16LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt16BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt32LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt32BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeFloatLE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个 32 位浮点数类型的值时，结果将是不确定的。 若参数 no和 offset 偏移量参数。 这意味着 value可能过大，或者 offs造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽
buf.writeFloatBE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个 32 位浮点数类型的值时，结果将是不确定的。 若参数 no和 offset 偏移量参数。 这意味着 value可能过大，或者 offs造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽
buf.writeDoubleLE(value,	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个有效的 64 位double 类型的值。 若参数 noAssert 为 true

offset[, noAssert])	参数。 这意味着 value 可能过大，或者 offset 可能会超出 b 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeDoubleBE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个有效的 64 位double 类型的值。 若参数 noAssert 为 true 参数。 这意味着 value 可能过大，或者 offset 可能会超出 b 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.fill(value[, offset][, end])	使用指定的 value 来填充这个 buffer。如果没有指定 offset (buffer.length)， 将会填充整个buffer。

Node.js Stream(流)

Stream 是一个抽象接口，Node 中有很多对象实现了这个接口。例如，对http 服务器发起请求的request 对象就是一个 Stream，还有stdout（标准输出）。

Node.js，Stream 有四种流类型：

- **Readable** - 可读操作。
- **Writable** - 可写操作。
- **Duplex** - 可读可写操作.
- **Transform** - 操作被写入数据，然后读出结果。

所有的 Stream 对象都是 EventEmitter 的实例。常用的事件有：

- **data** - 当有数据可读时触发。
- **end** - 没有更多的数据可读时触发。
- **error** - 在接收和写入过程中发生错误时触发。
- **finish** - 所有数据已被写入到底层系统时触发。

本教程会为大家介绍常用的流操作。

从流中读取数据

创建 input.txt 文件，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");
var data = '';

// 创建可读流
var readerStream = fs.createReadStream('input.txt');

// 设置编码为 utf8。
readerStream.setEncoding('UTF8');

// 处理流事件 --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function(){
   console.log(data);
});

readerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("程序执行完毕");
```

以上代码执行结果如下：

```
程序执行完毕
菜鸟教程官网地址：www.runoob.com
```

写入流

创建 main.js 文件, 代码如下：

```
var fs = require("fs");
var data = '菜鸟教程官网地址：www.runoob.com';

// 创建一个可以写入的流，写入到文件 output.txt 中
var writerStream = fs.createWriteStream('output.txt');

// 使用 utf8 编码写入数据
writerStream.write(data,'UTF8');

// 标记文件末尾
writerStream.end();

// 处理流事件 --> data, end, and error
writerStream.on('finish', function() {
    console.log("写入完成。");
});

writerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("程序执行完毕");
```

以上程序会将 data 变量的数据写入到 output.txt 文件中。代码执行结果如下：

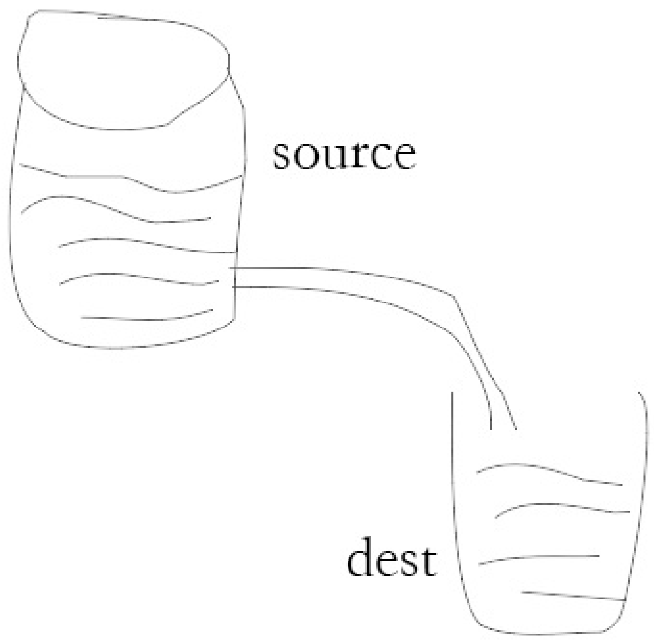
```
$ node main.js
程序执行完毕
写入完成。
```

查看 output.txt 文件的内容：

```
$ cat output.txt
菜鸟教程官网地址：www.runoob.com
```

管道流

管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。



如上面的图片所示，我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(pipe)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程。

以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。

设置 input.txt 文件内容如下：

```
菜鸟教程官网地址：www.runoob.com
管道流操作实例
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

// 创建一个可读流
var readerStream = fs.createReadStream('input.txt');

// 创建一个可写流
var writerStream = fs.createWriteStream('output.txt');

// 管道读写操作
// 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中
readerStream.pipe(writerStream);

console.log("程序执行完毕");
```

代码执行结果如下：

```
$ node main.js
程序执行完毕
```

查看 output.txt 文件的内容：

```
$ cat output.txt
菜鸟教程官网地址：www.runoob.com
管道流操作实例
```

链式流

链式是通过连接输出流到另外一个流并创建多个对个流操作链的机制。链式流一般用于管道操作。

接下来我们就是用管道和链式来压缩和解压文件。

创建 compress.js 文件, 代码如下：

```
var fs = require("fs");
var zlib = require('zlib');

// 压缩 input.txt 文件为 input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.gz'));

console.log("文件压缩完成。");
```

代码执行结果如下：

```
$ node compress.js
文件压缩完成。
```

执行完以上操作后，我们可以看到当前目录下生成了 input.txt 的压缩文件 input.txt.gz。

接下来，让我们来解压该文件，创建 decompress.js 文件，代码如下：

```
var fs = require("fs");
var zlib = require('zlib');

// 解压 input.txt.gz 文件为 input.txt
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input.txt'));

console.log("文件解压完成。");
```

代码执行结果如下：

```
$ node decompress.js
文件解压完成。
```

Node.js模块系统

为了让Node.js的文件可以相互调用，Node.js提供了一个简单的模块系统。

模块是Node.js 应用程序的基本组成部分，文件和模块是一一对应的。换言之，一个 Node.js 文件就是一个模块，这个文件可能是JavaScript 代码、JSON 或者编译过的C/C++ 扩展。

创建模块

在 Node.js 中，创建一个模块非常简单，如下我们创建一个 'main.js' 文件，代码如下:

```
var hello = require('./hello');
hello.world();
```

以上实例中，代码 require('./hello') 引入了当前目录下的hello.js文件（./ 为当前目录，node.js 默认后缀为js）。

Node.js 提供了exports 和 require 两个对象，其中 exports 是模块公开的接口，require 用于从外部获取一个模块的接口，即所获取模块的 exports 对象。

接下来我们就来创建hello.js文件，代码如下：

```
exports.world = function() {
  console.log('Hello World');
}
```

在以上示例中，hello.js 通过 exports 对象把 world 作为模块的访 问接口，在 main.js 中通过 require('./hello') 加载这个模块，然后就可以直接访 问main.js 中 exports 对象的成员函数了。

有时候我们只是想把一个对象封装到模块中，格式如下：

```
module.exports = function() {
  // ...
}
```

例如:

```
//hello.js
function Hello() {
    varname;
    this.setName = function(thyName) {
        name = thyName;
    };
    this.sayHello = function() {
        console.log('Hello ' + name);
    };
};
module.exports = Hello;
```

这样就可以直接获得这个对象了：

```
//main.js
var Hello = require('./hello');
hello = new Hello();
hello.setName('BYVoid');
hello.sayHello();
```

模块接口的唯一变化是使用 module.exports = Hello 代替了exports.world = function(){}。 在外部引用该模块时，其接口对象就是要输出的 Hello 对象本身，而不是原先的 exports。

服务端的模块放在哪里

也许你已经注意到，我们已经在代码中使用了模块了。像这样：

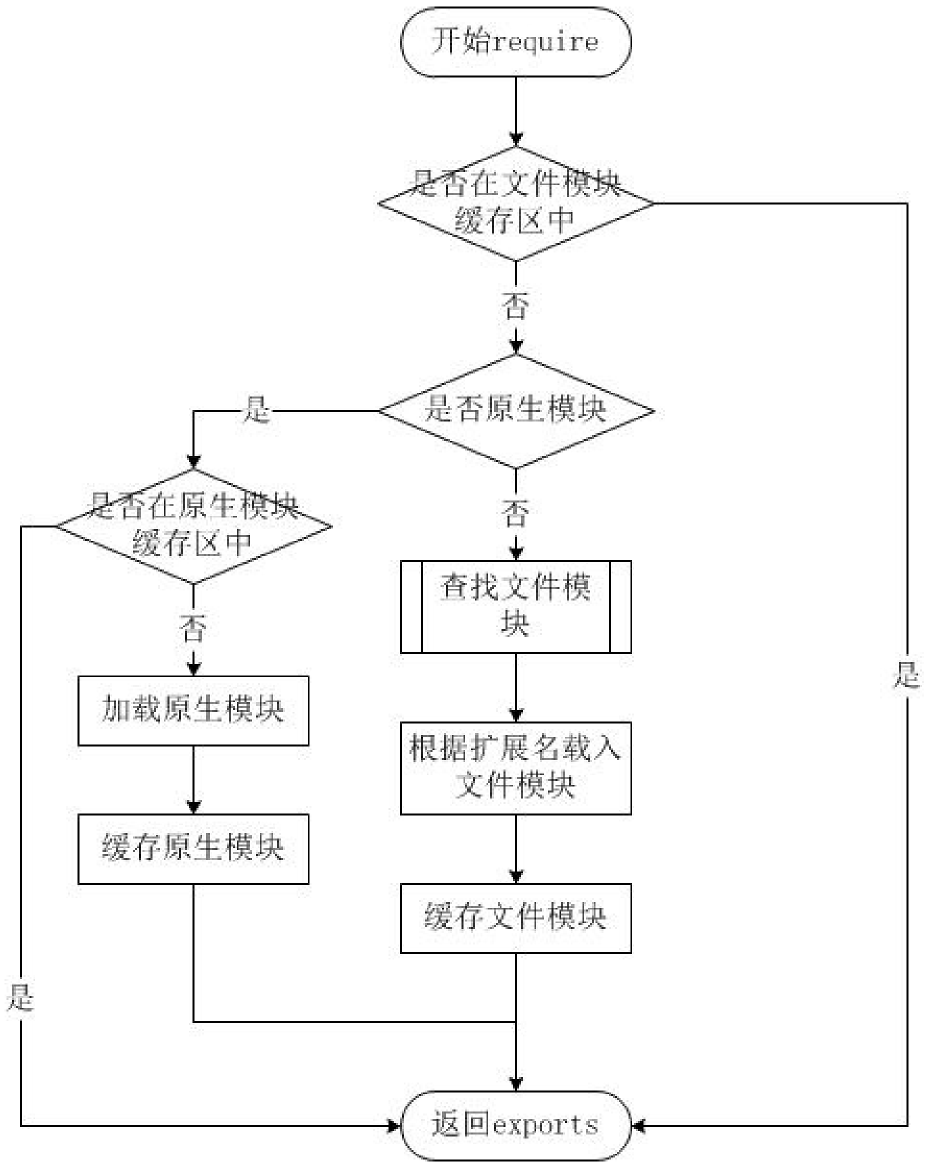
```
var http = require("http");

...

http.createServer(...);
```

Node.js中自带了一个叫做"http"的模块，我们在我们的代码中请求它并把返回值赋给一个本地变量。

这把我们的本地变量变成了一个拥有所有 http 模块所提供的公共方法的对象。

Node.js 的 require方法中的文件查找策略如下：

由于Node.js中存在4类模块（原生模块和3种文件模块），尽管require方法极其简单，但是内部的加载却是十分复杂的，其加载优先级也各自不同。如下图所示：



从文件模块缓存中加载

尽管原生模块与文件模块的优先级不同，但是都不会优先于从文件模块的缓存中加载已经存在的模块。

从原生模块加载

原生模块的优先级仅次于文件模块缓存的优先级。require方法在解析文件名之后，优先检查模块是否在原生模块列表中。以http模块为例，尽管在目录下存在一个http/http.js/http.node/http.json文件，require("http")都不会从这些文件中加载，而是从原生模块中加载。

原生模块也有一个缓存区，同样也是优先从缓存区加载。如果缓存区没有被加载过，则调用原生模块的加载方式进行加载和执行。

从文件加载

当文件模块缓存中不存在，而且不是原生模块的时候，Node.js会解析require方法传入的参数，并从文件系统中加载实际的文件，加载过程中的包装和编译细节在前一节中已经介绍过，这里我们将详细描述查找文件模块的过程，其中，也有一些细节值得知晓。

require方法接受以下几种参数的传递：

- http、fs、path等，原生模块。
- ./mod或../mod，相对路径的文件模块。
- /pathtomodule/mod，绝对路径的文件模块。
- mod，非原生模块的文件模块。

Node.js 函数

在JavaScript中，一个函数可以作为另一个函数接收一个参数。我们可以先定义一个函数，然后传递，也可以在传递参数的地方直接定义函数。

Node.js中函数的使用与Javascript类似，举例来说，你可以这样做：

```
function say(word) {
  console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

以上代码中，我们把 say 函数作为execute函数的第一个变量进行了传递。这里返回的不是 say 的返回值，而是 say 本身！

这样一来， say 就变成了execute 中的本地变量 someFunction ， execute可以通过调用 someFunction() （带括号的形式）来使用 say 函数。

当然，因为 say 有一个变量， execute 在调用 someFunction 时可以传递这样一个变量。

匿名函数

我们可以把一个函数作为变量传递。但是我们不一定要绕这个"先定义，再传递"的圈子，我们可以直接在另一个函数的括号中定义和传递这个函数：

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

我们在 execute 接受第一个参数的地方直接定义了我们准备传递给 execute 的函数。

用这种方式，我们甚至不用给这个函数起名字，这也是为什么它被叫做匿名函数 。

函数传递是如何让HTTP服务器工作的

带着这些知识，我们再来看看我们简约而不简单的HTTP服务器：

```
var http = require("http");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

现在它看上去应该清晰了很多：我们向 createServer 函数传递了一个匿名函数。

用这样的代码也可以达到同样的目的：

```
var http = require("http");

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);
```

Node.js 路由

我们要为路由提供请求的URL和其他需要的GET及POST参数，随后路由需要根据这些数据来执行相应的代码。

因此，我们需要查看HTTP请求，从中提取出请求的URL以及GET/POST参数。这一功能应当属于路由还是服务器（甚至作为一个模块自身的功能）确实值得探讨，但这里暂定其为我们的HTTP服务器的功能。

我们需要的所有数据都会包含在request对象中，该对象作为onRequest()回调函数的第一个参数传递。但是为了解析这些数据，我们需要额外的Node.JS模块，它们分别是url和querystring模块。

```
                     url.parse(string).query
                                           |
             url.parse(string).pathname      |
                         |                   |
                         |                   |
                       ------ -------------------
http://localhost:8888/start?foo=bar&hello=world
                                  ---       -----
                                   |          |
                                   |          |
                querystring(string)["foo"]    |
                                              |
                           querystring(string)["hello"]
```

当然我们也可以用querystring模块来解析POST请求体中的参数，稍后会有演示。

现在我们来给onRequest()函数加上一些逻辑，用来找出浏览器请求的URL路径：

```
var http = require("http");
var url = require("url");

function start() {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

好了，我们的应用现在可以通过请求的URL路径来区别不同请求了--这使我们得以使用路由（还未完成）来将请求以URL路径为基准映射到处理程序上。

在我们所要构建的应用中，这意味着来自/start和/upload的请求可以使用不同的代码来处理。稍后我们将看到这些内容是如何整合到一起的。

现在我们可以来编写路由了，建立一个名为router.js的文件，添加以下内容：

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

如你所见，这段代码什么也没干，不过对于现在来说这是应该的。在添加更多的逻辑以前，我们先来看看如何把路由和服务器整合起来。

我们的服务器应当知道路由的存在并加以有效利用。我们当然可以通过硬编码的方式将这一依赖项绑定到服务器上，但是其它语言的编程经验告诉我们这会是一件非常痛苦的事，因此我们将使用依赖注入的方式较松散地添加路由模块。

首先，我们来扩展一下服务器的start()函数，以便将路由函数作为参数传递过去：

```
var http = require("http");
var url = require("url");

function start(route) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

同时，我们会相应扩展index.js，使得路由函数可以被注入到服务器中：

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

在这里，我们传递的函数依旧什么也没做。

如果现在启动应用（node index.js，始终记得这个命令行），随后请求一个URL，你将会看到应用输出相应的信息，这表明我们的HTTP服务器已经在使用路由模块了，并会将请求的路径传递给路由：

```
bash$ node index.js
Request for /foo received.
About to route a request for /foo
```

以上输出已经去掉了比较烦人的/favicon.ico请求相关的部分。

Node.js 全局对象

JavaScript 中有一个特殊的对象，称为全局对象（Global Object），它及其所有属性都可 以在程序的任何地方访问，即全局变量。

在浏览器JavaScript 中，通常window 是全局对象， 而Node.js 中的全局对象是 global，所有全局变量（除了 global 本身以外）都是 global 对象的属性。

我们在Node.js 中能够直接访问到对象通常都是 global 的属性，如 console、process 等，下面逐一介绍。

全局对象与全局变量

global 最根本的作用是作为全局变量的宿主。按照ECMAScript 的定义，满足以下条 件的变量是全局变量：

- 在最外层定义的变量；
- 全局对象的属性；
- 隐式定义的变量（未定义直接赋值的变量）。

当你定义一个全局变量时，这个变量同时也会成为全局对象的属性，反之亦然。需要注 意的是，在Node.js 中你不可能在最外层定义变量，因为所有用户代码都是属于当前模块的， 而模块本身不是最外层上下文。

注意： 永远使用var 定义变量以避免引入全局变量，因为全局变量会污染 命名空间，提高代码的耦合风险。

process

process 是一个全局变量，即 global 对象的属性。

它用于描述当前Node.js 进程状态 的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要 和它打交道。下面将会介绍process 对象的一些最常用的成员方法。

process.argv是命令行参数数组，第一个元素是 node，第二个元素是脚本文件名， 从第三个元素开始每个元素是一个运行参数。

```
console.log(process.argv);
```

将以上代码存储为argv.js，通过以下命令运行：

```
$ node argv.js 1991 name=byvoid --v "Carbo Kuo"
[ 'node',
'/home/byvoid/argv.js',
'1991',
'name=byvoid',
'--v',
'Carbo Kuo' ]
```

- **process.stdout**是标准输出流，通常我们使用的 console.log() 向标准输出打印 字符，而 process.stdout.write() 函数提供了更底层的接口。
- **process.stdin**是标准输入流，初始时它是被暂停的，要想从标准输入读取数据， 你必须恢复流，并手动编写流的事件响应函数。

```
process.stdin.resume();
process.stdin.on('data', function(data) {
process.stdout.write('read from console: ' + data.toString());
});
```

- **process.nextTick(callback)**的功能是为事件循环设置一项任务，Node.js 会在 下次事件循环调响应时调用 callback。

初学者很可能不理解这个函数的作用，有什么任务不能在当下执行完，需要交给下次事 件循环响应来做呢？

我们讨论过，Node.js 适合I/O 密集型的应用，而不是计算密集型的应用， 因为一个Node.js 进程只有一个线程，因此在任何时刻都只有一个事件在执行。

如果这个事 件占用大量的CPU 时间，执行事件循环中的下一个事件就需要等待很久，因此 Node.js 的一 个编程原则就是尽量缩短每个事件的执行时间。process.nextTick() 提供了一个这样的 工具，可以把复杂的工作拆散， 变成一个个较小的事件。

```
functiondoSomething(args, callback) {
    somethingComplicated(args);
    callback();
}
doSomething(functiononEnd() {
    compute();
});
```

我们假设compute() 和somethingComplicated() 是两个较为耗时的函数，以上 的程序在调用 doSomething() 时会先执行somethingComplicated()，然后立即调用 回调函数，在 onEnd() 中又会执行 compute()。下面用process.nextTick() 改写上 面的程序：

```
functiondoSomething(args, callback) {
    somethingComplicated(args);
    process.nextTick(callback);
}
doSomething(functiononEnd() {
    compute();
});
```

改写后的程序会把上面耗时的操作拆分为两个事件，减少每个事件的执行时间，提高事 件响应速度。

注意： 不要使用setTimeout(fn,0)代替process.nextTick(callback)， 前者比后者效率要低得多。

我们探讨了process对象常用的几个成员，除此之外process还展示了process.platform、process.pid、process.execPath、process.memoryUsage() 等方法，以及POSIX 进程信号响应机制。有兴趣的读者可以访问http://nodejs.org/api/process.html 了解详细 内容。

console

console 用于提供控制台标准输出，它是由Internet Explorer 的JScript 引擎提供的调试 工具，后来逐渐成为浏览器的事实标准。

Node.js 沿用了这个标准，提供与习惯行为一致的 console 对象，用于向标准输出流（stdout）或标准错误流（stderr）输出字符。 ? console.log()：向标准输出流打印字符并以换行符结束。

console.log 接受若干 个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则 以类似于C 语言 printf() 命令的格式输出。

第一个参数是一个字符串，如果没有 参数，只打印一个换行。

```
console.log('Hello world');
console.log('byvoid%diovyb');
console.log('byvoid%diovyb', 1991);
```

运行结果为：

```
Hello world
byvoid%diovyb
byvoid1991iovyb
```

- console.error()：与console.log() 用法相同，只是向标准错误流输出。
- console.trace()：向标准错误流输出当前的调用栈。

```
console.trace();
```

运行结果为：

```
Trace:
at Object.<anonymous> (/home/byvoid/consoletrace.js:1:71)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

Node.js 常用工具 util

util 是一个Node.js 核心模块，提供常用函数的集合，用于弥补核心JavaScript 的功能 过于精简的不足。

util.inherits

util.inherits(constructor, superConstructor)是一个实现对象间原型继承 的函数。

JavaScript 的面向对象特性是基于原型的，与常见的基于类的不同。JavaScript 没有 提供对象继承的语言级别特性，而是通过原型复制来实现的。

在这里我们只介绍util.inherits 的用法，示例如下：

```
varutil = require('util');
functionBase() {
    this.name = 'base';
    this.base = 1991;
    this.sayHello = function() {
    console.log('Hello ' + this.name);
    };
}
Base.prototype.showName = function() {
    console.log(this.name);
};
functionSub() {
    this.name = 'sub';
}
util.inherits(Sub, Base);
varobjBase = newBase();
objBase.showName();
objBase.sayHello();
console.log(objBase);
varobjSub = newSub();
objSub.showName();
//objSub.sayHello();
console.log(objSub);
```

我们定义了一个基础对象Base 和一个继承自Base 的Sub，Base 有三个在构造函数 内定义的属性和一个原型中定义的函数，通过util.inherits 实现继承。运行结果如下：

```
base
Hello base
{ name: 'base', base: 1991, sayHello: [Function] }
sub
{ name: 'sub' }
```

注意：Sub 仅仅继承了Base 在原型中定义的函数，而构造函数内部创造的 base 属 性和 sayHello 函数都没有被 Sub 继承。

同时，在原型中定义的属性不会被console.log 作 为对象的属性输出。如果我们去掉 objSub.sayHello(); 这行的注释，将会看到：

```
node.js:201
throw e; // process.nextTick error, or 'error' event on first tick
^
TypeError: Object #<Sub> has no method 'sayHello'
at Object.<anonymous> (/home/byvoid/utilinherits.js:29:8)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

util.inspect

util.inspect(object,[showHidden],[depth],[colors])是一个将任意对象转换 为字符串的方法，通常用于调试和错误输出。它至少接受一个参数 object，即要转换的对象。

showHidden 是一个可选参数，如果值为 true，将会输出更多隐藏信息。

depth 表示最大递归的层数，如果对象很复杂，你可以指定层数以控制输出信息的多 少。如果不指定depth，默认会递归2层，指定为 null 表示将不限递归层数完整遍历对象。 如果color 值为 true，输出格式将会以ANSI 颜色编码，通常用于在终端显示更漂亮 的效果。

特别要指出的是，util.inspect 并不会简单地直接把对象转换为字符串，即使该对 象定义了 toString 方法也不会调用。

```
varutil = require('util');
functionPerson() {
    this.name = 'byvoid';
    this.toString = function() {
    return this.name;
    };
}
varobj = newPerson();
console.log(util.inspect(obj));
console.log(util.inspect(obj, true));
```

运行结果是：

```
{ name: 'byvoid', toString: [Function] }
{ toString:
{ [Function]
[prototype]: { [constructor]: [Circular] },
[caller]: null,
[length]: 0,
[name]: '',
[arguments]: null },
name: 'byvoid' }
```

util.isArray(object)

如果给定的参数 "object" 是一个数组返回true，否则返回false。

```
var util = require('util');

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

util.isRegExp(object)

如果给定的参数 "object" 是一个正则表达式返回true，否则返回false。

```
var util = require('util');

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

util.isDate(object)

如果给定的参数 "object" 是一个日期返回true，否则返回false。

```
var util = require('util');

util.isDate(new Date())
  // true
util.isDate(Date())
  // false (without 'new' returns a String)
util.isDate({})
  // false
```

util.isError(object)

如果给定的参数 "object" 是一个错误对象返回true，否则返回false。

```
var util = require('util');

util.isError(new Error())
  // true
util.isError(new TypeError())
  // true
util.isError({ name: 'Error', message: 'an error occurred' })
  // false
```

更多详情可以访问 http://nodejs.org/api/util.html 了解详细内容。

Node.js 文件系统

Node.js 提供一组类似 UNIX（POSIX）标准的文件操作API。 Node 导入文件系统模块(fs)语法如下所示：

```
var fs = require("fs")
```

异步和同步

Node.js 文件系统（fs 模块）模块中的方法均有异步和同步版本，例如读取文件内容的函数有异步的 fs.readFile() 和同步的 fs.readFileSync()。

异步的方法函数最后一个参数为回调函数，回调函数的第一个参数包含了错误信息(error)。

建议大家是用异步方法，比起同步，异步方法性能更高，速度更快，而且没有阻塞。

实例

创建 input.txt 文件，内容如下：

```
菜鸟教程官网地址：www.runoob.com
文件读取实例
```

创建 file.js 文件, 代码如下：

```
var fs = require("fs");

// 异步读取
fs.readFile('input.txt', function (err, data) {
   if (err) {
       return console.error(err);
   }
   console.log("异步读取: " + data.toString());
});

// 同步读取
var data = fs.readFileSync('input.txt');
console.log("同步读取: " + data.toString());

console.log("程序执行完毕。");
```

以上代码执行结果如下：

```
$ node file.js
同步读取: 菜鸟教程官网地址：www.runoob.com
文件读取实例

程序执行完毕。
异步读取: 菜鸟教程官网地址：www.runoob.com
文件读取实例
```

接下来，让我们来具体了解下 Node.js 文件系统的方法。

打开文件

语法

以下为在异步模式下打开文件的语法格式：

```
fs.open(path, flags[, mode], callback)
```

参数

参数使用说明如下：

- **path** - 文件的路径。
- **flags** - 文件打开的行为。具体值详见下文。
- **mode** - 设置文件模式(权限)，文件创建默认权限为 0666(可读，可写)。
- **callback** - 回调函数，带有两个参数如：callback(err, fd)。

flags 参数可以是以下值：

Flag	描述
r	以读取模式打开文件。如果文件不存在抛出异常。
r+	以读写模式打开文件。如果文件不存在抛出异常。
rs	以同步的方式读取文件。
rs+	以同步的方式读取和写入文件。
w	以写入模式打开文件，如果文件不存在则创建。
wx	类似 'w'，但是如果文件路径不存在，则文件写入失败。
w+	以读写模式打开文件，如果文件不存在则创建。
wx+	类似 'w+'， 但是如果文件路径不存在，则文件读写失败。
a	以追加模式打开文件，如果文件不存在则创建。
ax	类似 'a'， 但是如果文件路径不存在，则文件追加失败。
a+	以读取追加模式打开文件，如果文件不存在则创建。
ax+	类似 'a+'， 但是如果文件路径不存在，则文件读取追加失败。

实例

接下来我们创建 file.js 文件，并打开 input.txt 文件进行读写，代码如下所示：

```
var fs = require("fs");

// 异步打开文件
console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
  console.log("文件打开成功！");
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
```

获取文件信息

语法

以下为通过异步模式获取文件信息的语法格式：

```
fs.stat(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，带有两个参数如：(err, stats), **stats** 是 fs.Stats 对象。

fs.stat(path)执行后，会将stats类的实例返回给其回调函数。可以通过stats类中的提供方法判断文件的相关属性。例如判断是否为文件：

```
var fs = require('fs');

fs.stat('/Users/liuht/code/itbilu/demo/fs.js', function (err, stats) {
    console.log(stats.isFile());         //true
})
```

stats类中的方法有：

方法	描述
stats.isFile()	如果是文件返回 true，否则返回 false。
stats.isDirectory()	如果是目录返回 true，否则返回 false。
stats.isBlockDevice()	如果是块设备返回 true，否则返回 false。
stats.isCharacterDevice()	如果是字符设备返回 true，否则返回 false。
stats.isSymbolicLink()	如果是软链接返回 true，否则返回 false。
stats.isFIFO()	如果是FIFO，返回true，否则返回 false。FIFO是UNIX中的一种特殊类型的命令管道。
stats.isSocket()	如果是 Socket 返回 true，否则返回 false。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备打开文件！");
fs.stat('input.txt', function (err, stats) {
   if (err) {
       return console.error(err);
   }
   console.log(stats);
   console.log("读取文件信息成功！");

   // 检测文件类型
   console.log("是否为文件(isFile) ? " + stats.isFile());
   console.log("是否为目录(isDirectory) ? " + stats.isDirectory());
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
{ dev: 16777220,
  mode: 33188,
  nlink: 1,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 40333161,
  size: 61,
  blocks: 8,
  atime: Mon Sep 07 2015 17:43:55 GMT+0800 (CST),
  mtime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST),
  ctime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST) }
读取文件信息成功！
是否为文件(isFile) ? true
是否为目录(isDirectory) ? false
```

写入文件

语法

以下为异步模式下写入文件的语法格式：

```
fs.writeFile(filename, data[, options], callback)
```

如果文件存在，该方法写入的内容会覆盖旧的文件内容。

参数

参数使用说明如下：

- **path** - 文件路径。
- **data** - 要写入文件的数据，可以是 String(字符串) 或 Buffer(流) 对象。

- **options** - 该参数是一个对象，包含 {encoding, mode, flag}。默认编码为 utf8, 模式为 0666 ， flag 为 'w'
- **callback** - 回调函数，回调函数只包含错误信息参数(err)，在写入失败时返回。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备写入文件");
fs.writeFile('input.txt', '我是通过写入的文件内容！',  function(err) {
   if (err) {
       return console.error(err);
   }
   console.log("数据写入成功！");
   console.log("--------我是分割线-------------")
   console.log("读取写入的数据！");
   fs.readFile('input.txt', function (err, data) {
      if (err) {
         return console.error(err);
      }
      console.log("异步读取文件数据: " + data.toString());
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备写入文件
数据写入成功！
--------我是分割线-------------
读取写入的数据！
异步读取文件数据: 我是通过写入的文件内容
```

读取文件

语法

以下为异步模式下读取文件的语法格式：

```
fs.read(fd, buffer, offset, length, position, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **buffer** - 数据写入的缓冲区。
- **offset** - 缓冲区写入的写入偏移量。
- **length** - 要从文件中读取的字节数。
- **position** - 文件读取的起始位置，如果 position 的值为 null，则会从当前文件指针的位置读取。
- **callback** - 回调函数，有三个参数err, bytesRead, buffer，err 为错误信息， bytesRead 表示读取的字节数，buffer 为缓冲区对象。

实例

input.txt 文件内容为：

```
菜鸟教程官网地址：www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开已存在的文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("准备读取文件：");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }
      console.log(bytes + "  字节被读取");

      // 仅输出读取的字节
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开已存在的文件！
文件打开成功！
准备读取文件：
42  字节被读取
菜鸟教程官网地址：www.runoob.com
```

关闭文件

语法

以下为异步模式下关闭文件的语法格式：

```
fs.close(fd, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
菜鸟教程官网地址：www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("准备读取文件！");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }

      // 仅输出读取的字节
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }

      // 关闭文件
      fs.close(fd, function(err){
         if (err){
            console.log(err);
         }
         console.log("文件关闭成功");
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
准备读取文件！
菜鸟教程官网地址：www.runoob.com
文件关闭成功
```

截取文件

语法

以下为异步模式下截取文件的语法格式：

```
fs.ftruncate(fd, len, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。

- **len** - 文件内容截取的长度。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("截取10字节后的文件内容。");

   // 截取文件
   fs.ftruncate(fd, 10, function(err){
      if (err){
         console.log(err);
      }
      console.log("文件截取成功。");
      console.log("读取相同的文件");
      fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
         if (err){
            console.log(err);
         }

         // 仅输出读取的字节
         if(bytes > 0){
            console.log(buf.slice(0, bytes).toString());
         }

         // 关闭文件
         fs.close(fd, function(err){
            if (err){
               console.log(err);
            }
            console.log("文件关闭成功！");
         });
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
截取10字节后的文件内容。
文件截取成功。
读取相同的文件
site:www.r
文件关闭成功
```

删除文件

语法

以下为删除文件的语法格式：

```
fs.unlink(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备删除文件！");
fs.unlink('input.txt', function(err) {
   if (err) {
       return console.error(err);
   }
   console.log("文件删除成功！");
});
```

以上代码执行结果如下：

```
$ node file.js
准备删除文件！
文件删除成功！
```

再去查看 input.txt 文件，发现已经不存在了。

创建目录

语法

以下为创建目录的语法格式：

```
fs.mkdir(path[, mode], callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **mode** - 设置目录权限，默认为 0777。
- **callback** - 回调函数，没有参数。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log(创建目录 /tmp/test");
fs.mkdir('/tmp/test',function(err){
   if (err) {
       return console.error(err);
   }
   console.log("目录创建成功。");
});
```

以上代码执行结果如下：

```
$ node file.js
创建目录 /tmp/test
目录创建成功。
```

读取目录

语法

以下为读取目录的语法格式：

```
fs.readdir(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，回调函数带有两个参数err, files，err 为错误信息，files 为 目录下的文件数组列表。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("查看 /tmp 目录");
fs.readdir("/tmp/",function(err, files){
   if (err) {
       return console.error(err);
   }
   files.forEach( function (file){
       console.log( file );
   });
});
```

以上代码执行结果如下：

```
$ node file.js
查看 /tmp 目录
input.out
output.out
test
test.txt
```

删除目录

语法

以下为删除目录的语法格式：

```
fs.rmdir(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备删除目录 /tmp/test");
fs.rmdir("/tmp/test",function(err){
   if (err) {
       return console.error(err);
   }
   console.log("读取 /tmp 目录");
   fs.readdir("/tmp/",function(err, files){
      if (err) {
          return console.error(err);
      }
      files.forEach( function (file){
          console.log( file );
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备删除目录 /tmp/test
input.out
output.out
test
test.txt
读取 /tmp 目录
……
```

文件模块方法参考手册

以下为 Node.js 文件模块相同的方法列表：

方法	描述
fs.rename(oldPath, newPath, callback)	异步 rename().回调函数没有参数，但可能抛出异常。
fs.ftruncate(fd, len, callback)	异步 ftruncate().回调函数没有参数，但可能抛出异常。
fs.ftruncateSync(fd, len)	同步 ftruncate()

fs.truncate(path, len, callback)	异步 truncate().回调函数没有参数，但可能抛出异常。
fs.truncateSync(path, len)	同步 truncate()
fs.chown(path, uid, gid, callback)	异步 chown().回调函数没有参数，但可能抛出异常。
fs.chownSync(path, uid, gid)	同步 chown()
fs.fchown(fd, uid, gid, callback)	异步 fchown().回调函数没有参数，但可能抛出异常。
fs.fchownSync(fd, uid, gid)	同步 fchown()
fs.lchown(path, uid, gid, callback)	异步 lchown().回调函数没有参数，但可能抛出异常。
fs.lchownSync(path, uid, gid)	同步 lchown()
fs.chmod(path, mode, callback)	异步 chmod().回调函数没有参数，但可能抛出异常。
fs.chmodSync(path, mode)	同步 chmod().
fs.fchmod(fd, mode, callback)	异步 fchmod().回调函数没有参数，但可能抛出异常。
fs.fchmodSync(fd, mode)	同步 fchmod().
fs.lchmod(path, mode, callback)	异步 lchmod().回调函数没有参数，但可能抛出异常。Only available on Mac OS X.
fs.lchmodSync(path, mode)	同步 lchmod().
fs.stat(path, callback)	异步 stat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.lstat(path, callback)	异步 lstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.fstat(fd, callback)	异步 fstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.statSync(path)	同步 stat(). 返回 fs.Stats 的实例。
fs.lstatSync(path)	同步 lstat(). 返回 fs.Stats 的实例。
fs.fstatSync(fd)	同步 fstat(). 返回 fs.Stats 的实例。
fs.link(srcpath, dstpath, callback)	异步 link().回调函数没有参数，但可能抛出异常。
fs.linkSync(srcpath, dstpath)	同步 link().
fs.symlink(srcpath, dstpath[, type], callback)	异步 symlink().回调函数没有参数，但可能抛出异常。type 参数可以设置为 'dir', 'file', 或 'junction' (默认为

dstpath[, type], callback)	'file') 。
fs.symlinkSync(srcpath, dstpath[, type])	同步 symlink().
fs.readlink(path, callback)	异步 readlink(). 回调函数有两个参数 err, linkString。
fs.realpath(path[, cache], callback)	异步 realpath(). 回调函数有两个参数 err, resolvedPath。
fs.realpathSync(path[, cache])	同步 realpath()。返回绝对路径。
fs.unlink(path, callback)	异步 unlink().回调函数没有参数，但可能抛出异常。
fs.unlinkSync(path)	同步 unlink().
fs.rmdir(path, callback)	异步 rmdir().回调函数没有参数，但可能抛出异常。
fs.rmdirSync(path)	同步 rmdir().
fs.mkdir(path[, mode], callback)	S异步 mkdir(2).回调函数没有参数，但可能抛出异常。 mode defaults to 0777.
fs.mkdirSync(path[, mode])	同步 mkdir().
fs.readdir(path, callback)	异步 readdir(3). 读取目录的内容。
fs.readdirSync(path)	同步 readdir().返回文件数组列表。
fs.close(fd, callback)	异步 close().回调函数没有参数，但可能抛出异常。
fs.closeSync(fd)	同步 close().
fs.open(path, flags[, mode], callback)	异步打开文件。
fs.openSync(path, flags[, mode])	同步 version of fs.open().
fs.utimes(path, atime, mtime, callback)	?
fs.utimesSync(path, atime, mtime)	修改文件时间戳，文件通过指定的文件路径。
fs.futimes(fd, atime, mtime, callback)	?
fs.futimesSync(fd, atime, mtime)	修改文件时间戳，通过文件描述符指定。
fs.fsync(fd, callback)	异步 fsync.回调函数没有参数，但可能抛出异常。
fs.fsyncSync(fd)	同步 fsync.
fs.write(fd, buffer, offset, length[, position], callback)	将缓冲区内容写入到通过文件描述符指定的文件。
fs.write(fd, data[, position[,	

encoding]], callback)	
fs.writeSync(fd, buffer, offset, length[, position])	同步版的 fs.write()。
fs.writeSync(fd, data[, position[, encoding]])	同步版的 fs.write().
fs.read(fd, buffer, offset, length, position, callback)	通过文件描述符 fd 读取文件内容。
fs.readSync(fd, buffer, offset, length, position)	同步版的 fs.read.
fs.readFile(filename[, options], callback)	异步读取文件内容。
fs.readFileSync(filename[, options])	
fs.writeFile(filename, data[, options], callback)	异步写入文件内容。
fs.writeFileSync(filename, data[, options])	同步版的 fs.writeFile。
fs.appendFile(filename, data[, options], callback)	异步追加文件内容。
fs.appendFileSync(filename, data[, options])	The 同步 version of fs.appendFile.
fs.watchFile(filename[, options], listener)	查看文件的修改。
fs.unwatchFile(filename[, listener])	停止查看 filename 的修改。
fs.watch(filename[, options][, listener])	查看 filename 的修改，filename 可以是文件或目录。返回 fs.FSWatcher 对象。
fs.exists(path, callback)	检测给定的路径是否存在。
fs.existsSync(path)	同步版的 fs.exists.
fs.access(path[, mode], callback)	测试指定路径用户权限。
fs.accessSync(path[, mode])	同步版的 fs.access。
fs.createReadStream(path[, options])	返回ReadStream 对象。
fs.createWriteStream(path[, options])	返回 WriteStream 对象。
fs.symlink(srcpath, dstpath[, type], callback)	异步 symlink().回调函数没有参数，但可能抛出异常。

dstpath[, type], callback)	

更多内容，请查看官网文件模块描述：File System。

Node.js GET/POST请求

在很多场景中，我们的服务器都需要跟用户的浏览器打交道，如表单提交。

表单提交到服务器一般都使用GET/POST请求。

本章节我们将为大家介绍 Node.js GET/POST请求。

获取GET请求内容

由于GET请求直接被嵌入在路径中，URL是完整的请求路径，包括了?后面的部分，因此你可以手动解析后面的内容作为GET请求的参数。

node.js中url模块中的parse函数提供了这个功能。

```
var http = require('http');
var url = require('url');
var util = require('util');

http.createServer(function(req, res){
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end(util.inspect(url.parse(req.url, true)));
}).listen(3000);
```

在浏览器中访问 `http://localhost:3000/user?name=w3c&email=w3c@w3cschool.cc` 然后查看返回结果:

```
{ protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=w3c&email=w3c@w3cschool.cc',
  query: { name: 'w3c', email: 'w3c@w3cschool.cc' },
  pathname: '/user',
  path: '/user?name=w3c&email=w3c@w3cschool.cc',
  href: '/user?name=w3c&email=w3c@w3cschool.cc' }
```

获取POST请求内容

POST请求的内容全部的都在请求体中，http.ServerRequest并没有一个属性内容为请求体，原因是等待请求体传输可能是一件耗时的工作。

比如上传文件，而很多时候我们可能并不需要理会请求体的内容，恶意的POST请求会大大消耗服务器的资源，所有node.js默认是不会解析请求体的， 当你需要的时候，需要手动来做。

```
var http = require('http');
var querystring = require('querystring');
var util = require('util');

http.createServer(function(req, res){
    var post = '';     //定义了一个post变量，用于暂存请求体的信息

    req.on('data', function(chunk){    //通过req的data事件监听函数，每当接受到请求体的数据，就累加
        post += chunk;
    });

    req.on('end', function(){    //在end事件触发后，通过querystring.parse将post解析为真正的POST
        post = querystring.parse(post);
        res.end(util.inspect(post));
    });
}).listen(3000);
```

Node.js 工具模块

在 Node.js 模块库中有很多好用的模块。接下来我们为大家介绍几种常用模块的使用：

模块名	描述
OS 模块	提供基本的系统操作函数。
Path 模块	提供了处理和转换文件路的工具。
Net 模块	用于底层的网络通信。提供了服务端和客户端的的操作。
DNS 模块	用于解析域名。
Domain 模块	简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的。

Node.js OS 模块

Node.js os 模块提供了一些基本的系统操作函数。我们可以通过以下方式引入该模块：

```
var os =  require("os")
```

方法

方法	描述
os.tmpdir()	返回操作系统的默认临时文件夹。
os.endianness()	返回 CPU 的字节序，可能的是 "BE" 或 "LE"。
os.hostname()	返回操作系统的主机名。
os.type()	返回操作系统名
os.platform()	返回操作系统名
os.arch()	返回操作系统 CPU 架构，可能的值有 "x64"、"arm" 和 "ia32"。
os.release()	返回操作系统的发行版本。
os.uptime()	返回操作系统运行的时间，以秒为单位。
os.loadavg()	返回一个包含 1、5、15 分钟平均负载的数组。
os.totalmem()	返回系统内存总量，单位为字节。
os.freemem()	返回操作系统空闲内存量，单位是字节。
os.cpus()	返回一个对象数组，包含所安装的每个 CPU/内核的信息：型号、速度（单位 MHz）、时间（一个包含 user、nice、sys、idle 和 irq 所使用 CPU/内核毫秒数的对象）。
os.networkInterfaces()	获得网络接口列表。

属性

属性	描述
os.EOL	定义了操作系统的行尾符的常量。

实例

创建 main.js 文件，代码如下所示：

```
var os =  require("os");  // CPU 的字节序 console.log('endianness : '  + os.endianness());
```

代码执行结果如下：

```
$ node main.js
endianness : LE
type :  Linux platform : linux
total memory :  25103400960 bytes. free memory :  20676710400 bytes.
```

Node.js Path 模块

Node.js path 模块提供了一些用于处理文件路径的小工具，我们可以通过以下方式引入该模块：

```
var path = require("path")
```

方法

方法	描述
path.normalize(p)	规范化路径，注意'..' 和 '.'。
path.join([path1][, path2][, ...])	用于连接路径。该方法的主要用途在于，会正确使用当前系统的路径分隔符，Unix系统是"/"，Windows系统是"\"。
path.resolve([from ...], to)	将 **to** 参数解析为绝对路径。
path.isAbsolute(path)	判断参数 **path** 是否是绝对路径。
path.relative(from, to)	用于将相对路径转为绝对路径。
path.dirname(p)	返回路径中代表文件夹的部分，同 Unix 的dirname 命令类似。
path.basename(p[, ext])	返回路径中的最后一部分。同 Unix 命令 bashname 类似。
path.extname(p)	返回路径中文件的后缀名，即路径中最后一个'.'之后的部分。如果一个路径中并不包含'.'或该路径只包含一个'.' 且这个'.'为路径的第一个字符，则此命令返回空字符串。
path.parse(pathString)	返回路径字符串的对象。
path.format(pathObject)	从对象中返回路径字符串，和 path.parse 相反。

属性

属性	描述
path.sep	平台的文件路径分隔符，'\' 或 '/'。
path.delimiter	平台的分隔符, ; or ':'.
path.posix	提供上述 path 的方法，不过总是以 posix 兼容的方式交互。
path.win32	提供上述 path 的方法，不过总是以 win32 兼容的方式交互。

实例

创建 main.js 文件，代码如下所示：

```
var path = require("path");

// 格式化路径
console.log('normalization : ' + path.normalize('/test/test1//2slashes/1slash/tab/..'));

// 连接路径
console.log('joint path : ' + path.join('/test', 'test1', '2slashes/1slash', 'tab', '..')

// 转换为绝对路径
console.log('resolve : ' + path.resolve('main.js'));

// 路径中文件的后缀名
console.log('ext name : ' + path.extname('main.js'));
```

代码执行结果如下：

```
$ node main.js
normalization : /test/test1/2slashes/1slash
joint path : /test/test1/2slashes/1slash
resolve : /web/com/1427176256_27423/main.js
ext name : .js
```

Node.js Net 模块

Node.js Net 模块提供了一些用于底层的网络通信的小工具，包含了创建服务器/客户端的方法，我们可以通过以下方式引入该模块：

```
var net = require("net")
```

方法

方法	描述
net.createServer([options][, connectionListener])	创建一个 TCP 服务器。参数 connectionListener 自动给 'connection' 事件创建监听器。
net.connect(options[, connectionListener])	返回一个新的 'net.Socket'，并连接到指定的地址和端口。 当 socket 建立的时候，将会触发 'connect' 事件。
net.createConnection(options[, connectionListener])	创建一个到端口 port 和 主机 host的 TCP 连接。 host 默认为 'localhost'。
net.connect(port[, host][, connectListener])	创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.createConnection(port[, host][, connectListener])	创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.connect(path[, connectListener])	创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件上。返回 'net.Socket'。
net.createConnection(path[, connectListener])	创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.isIP(input)	检测输入的是否为 IP 地址。 IPV4 返回 4， IPV6 返回 6，其他情况返回 0。
net.isIPv4(input)	如果输入的地址为 IPV4， 返回 true，否则返回 false。
net.isIPv6(input)	如果输入的地址为 IPV6， 返回 true，否则返回 false。

net.Server

net.Server通常用于创建一个 TCP 或本地服务器。

方法	描述
server.listen(port[, host][, backlog][, callback])	监听指定端口 port 和 主机 host ac连接。 默认情况下 host 接受任何 IPv4 地址(INADDR_ANY)的直接连接。端口 port 为 0 时，则会分配一个随机端口。
server.listen(path[, callback])	通过指定 path 的连接， 启动一个本地 socket 服务器。
server.listen(handle[, callback])	通过指定句柄连接。
server.listen(options[, callback])	options 的属性：端口 port, 主机 host, 和 backlog, 以及可选参数 callback 函数, 他们在一起调用server.listen(port, [host], [backlog], [callback])。还有， 参数 path 可以用来指定 UNIX socket。
server.close([callback])	服务器停止接收新的连接，保持现有连接。这是异步函数， 当所有连接结束的时候服务器会关闭，并会触发 'close' 事件。
server.address()	操作系统返回绑定的地址，协议族名和服务器端口。
server.unref()	如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。
server.ref()	与 unref 相反， 如果这是唯一的服务器， 在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref，则再次调用 ref 并不会产生影响。
server.getConnections(callback)	异步获取服务器当前活跃连接的数量。当 socket 发送给子进程后才有效；回调函数有 2 个参数 err 和 count。

事件

事件	描述
listening	当服务器调用 server.listen 绑定后会触发。
connection	当新连接创建后会被触发。socket 是 net.Socket实例。
close	服务器关闭时会触发。注意，如果存在连接，这个事件不会被触发直到所有的连接关闭。
error	发生错误时触发。'close' 事件将被下列事件直接调用。

net.Socket

net.Socket 对象是 TCP 或 UNIX Socket 的抽象。net.Socket 实例实现了一个双工流接口。他们可以在用户创建客户端(使用 connect())时使用, 或者由 Node 创建它们，并通过 connection 服务器事件传递给用户。

事件

net.Socket 事件有：

事件	描述
lookup	在解析域名后，但在连接前，触发这个事件。对 UNIX sokcet 不适用。
connect	成功建立 socket 连接时触发。
data	当接收到数据时触发。
end	当 socket 另一端发送 FIN 包时，触发该事件。
timeout	当 socket 空闲超时时触发，仅是表明 socket 已经空闲。用户必须手动关闭连接。
drain	当写缓存为空得时候触发。可用来控制上传。
error	错误发生时触发。
close	当 socket 完全关闭时触发。参数 had_error 是布尔值，它表示是否因为传输错误导致 socket 关闭。

属性

net.Socket 提供了很多有用的属性，便于控制 socket 交互：

属性	描述
socket.bufferSize	该属性显示了要写入缓冲区的字节数。
socket.remoteAddress	远程的 IP 地址字符串，例如：'74.125.127.100' or '2001:4860:a005::68'。
socket.remoteFamily	远程IP协议族字符串，比如 'IPv4' or 'IPv6'。
socket.remotePort	远程端口，数字表示，例如：80 or 21。
socket.localAddress	网络连接绑定的本地接口 远程客户端正在连接的本地 IP 地址，字符串表示。例如，如果你在监听'0.0.0.0'而客户端连接在'192.168.1.1'，这个值就会是 '192.168.1.1'。
socket.localPort	本地端口地址，数字表示。例如：80 or 21。
socket.bytesRead	接收到得字节数。
socket.bytesWritten	发送的字节数。

方法

方法	描述
new net.Socket([options])	构造一个新的 socket 对象。
socket.connect(port[, host][, connectListener])	指定端口 port 和 主机 host，创建 socket 连接 。参数 host 默认为 localhost。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。
socket.connect(path[, connectListener])	打开指定路径的 unix socket。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。
socket.setEncoding([encoding])	设置编码
socket.write(data[, encoding][, callback])	在 socket 上发送数据。第二个参数指定了字符串的编码，默认是 UTF8 编码。
socket.end([data][, encoding])	半关闭 socket。例如，它发送一个 FIN 包。可能服务器仍在发送数据。
socket.destroy()	确保没有 I/O 活动在这个套接字上。只有在错误发生情况下才需要。（处理错误等等）。
socket.pause()	暂停读取数据。就是说，不会再触发 data 事件。对于控制上传非常有用。
socket.resume()	调用 pause() 后想恢复读取数据。
socket.setTimeout(timeout[, callback])	socket 闲置时间超过 timeout 毫秒后 ，将 socket 设置为超时。
socket.setNoDelay([noDelay])	禁用纳格（Nagle）算法。默认情况下 TCP 连接使用纳格算法，在发送前他们会缓冲数据。将 noDelay 设置为 true 将会在调用 socket.write() 时立即发送数据。noDelay 默认值为 true。
socket.setKeepAlive([enable][, initialDelay])	禁用/启用长连接功能，并在发送第一个在闲置 socket 上的长连接 probe 之前，可选地设定初始延时。默认为 false。 设定 initialDelay （毫秒），来设定收到的最后一个数据包和第一个长连接 probe之间的延时。将 initialDelay 设为0，将会保留默认（或者之前）的值。默认值为0.
socket.address()	操作系统返回绑定的地址，协议族名和服务器端口。返回的对象有 3 个属性，比如{ port: 12346, family: 'IPv4', address: '127.0.0.1' }。
socket.unref()	如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。如果服务器已被 unref，则再次调用 unref 并不会产生影响。
socket.ref()	与 unref 相反，如果这是唯一的服务器，在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref，则再次调用 ref 并不会产生影响。

实例

创建 server.js 文件，代码如下所示：

```
var net = require('net');
var server = net.createServer(function(connection) {
   console.log('client connected');
   connection.on('end', function() {
      console.log('客户端关闭连接');
   });
   connection.write('Hello World!\r\n');
   connection.pipe(connection);
});
server.listen(8080, function() {
  console.log('server is listening');
});
```

执行以上服务端代码：

```
$ node server.js
server is listening   # 服务已创建并监听 8080 端口
```

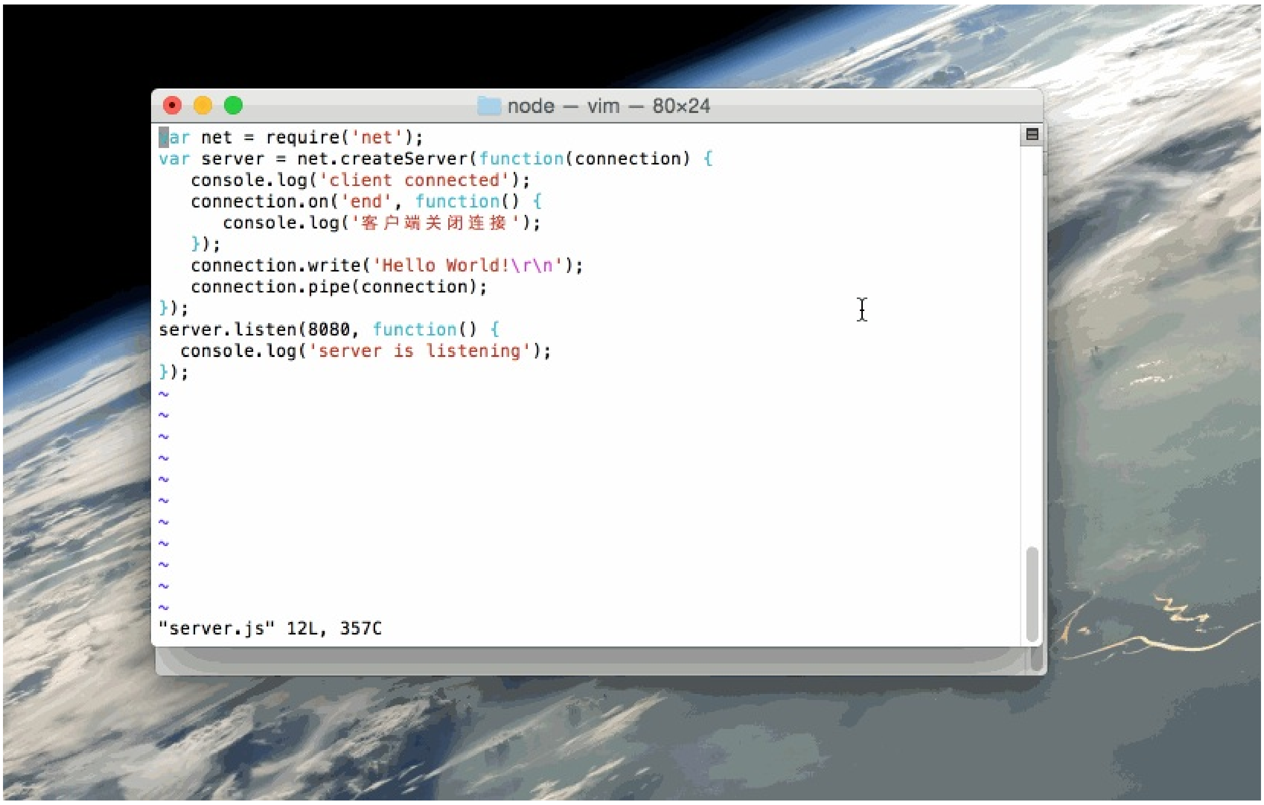
新开一个窗口，创建 client.js 文件，代码如下所示：

```
var net = require('net');
var client = net.connect({port: 8080}, function() {
   console.log('连接到服务器！');
});
client.on('data', function(data) {
   console.log(data.toString());
   client.end();
});
client.on('end', function() {
   console.log('断开与服务器的连接');
});
```

执行以上客户端的代码：

```
连接到服务器！
Hello World!

断开与服务器的连接
```

Gif 实例演示


node — vim — 80×24
var net = require('net');
var server = net.createServer(function(connection) {
console.log('client connected');
connection.on('end', function() {
console.log('客户端关闭连接');
});
connection.write('Hello World!\r\n');
connection.pipe(connection);
});
server.listen(8080, function() {
console.log('server is listening');
});
"server.js" 12L, 357C

Node.js DNS 模块

Node.js **DNS** 模块用于解析域名。引入 DNS 模块语法格式如下：

```
var dns = require("dns")
```

方法

方法	描述
dns.lookup(hostname[, options], callback)	将域名（比如 'runoob.com'）解析为第一条找到的记录 A （IPV4）或 AAAA(IPV6)。参数 options可以是一个对象或整数。如果没有提供 options，IP v4 和 v6 地址都可以。如果 options 是整数，则必须是 4 或 6。
dns.lookupService(address, port, callback)	使用 getnameinfo 解析传入的地址和端口为域名和服务。
dns.resolve(hostname[, rrtype], callback)	将一个域名（如 'runoob.com'）解析为一个 rrtype 指定记录类型的数组。
dns.resolve4(hostname, callback)	和 dns.resolve() 类似, 仅能查询 IPv4 (A 记录）。 addresses IPv4 地址数组 (比如，['74.125.79.104', '74.125.79.105', '74.125.79.106']）。
dns.resolve6(hostname, callback)	和 dns.resolve4() 类似， 仅能查询 IPv6(AAAA 查询）
dns.resolveMx(hostname, callback)	和 dns.resolve() 类似, 仅能查询邮件交换(MX 记录)。
dns.resolveTxt(hostname, callback)	和 dns.resolve() 类似, 仅能进行文本查询 (TXT 记录）。 addresses 是 2-d 文本记录数组。(比如，[['v=spf1 ip4:0.0.0.0 ', '~all']]）。 每个子数组包含一条记录的 TXT 块。根据使用情况可以连接在一起，也可单独使用。
dns.resolveSrv(hostname, callback)	和 dns.resolve() 类似, 仅能进行服务记录查询 (SRV 记录）。 addresses 是 hostname可用的 SRV 记录数组。 SRV 记录属性有优先级（priority）， 权重（weight）, 端口（port）, 和名字（name） (比如，[{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]）。
dns.resolveSoa(hostname, callback)	和 dns.resolve() 类似, 仅能查询权威记录(SOA 记录）。
dns.resolveNs(hostname, callback)	和 dns.resolve() 类似, 仅能进行域名服务器记录查询(NS 记录）。 addresses 是域名服务器记录数组（hostname 可以使用） (比如, ['ns1.example.com', 'ns2.example.com']）。
dns.resolveCname(hostname, callback)	和 dns.resolve() 类似, 仅能进行别名记录查询 (CNAME记录)。addresses 是对 hostname 可用的别名记录数组 (比如，, ['bar.example.com']）。
dns.reverse(ip, callback)	反向解析 IP 地址，指向该 IP 地址的域名数组。
dns.getServers()	返回一个用于当前解析的 IP 地址数组的字符串。
dns.setServers(servers)	指定一组 IP 地址作为解析服务器。

rrtypes

以下列出了 dns.resolve() 方法中有效的 rrtypes值:

- `'A'` IPV4 地址, 默认
- `'AAAA'` IPV6 地址
- `'MX'` 邮件交换记录
- `'TXT'` text 记录
- `'SRV'` SRV 记录
- `'PTR'` 用来反向 IP 查找
- `'NS'` 域名服务器记录
- `'CNAME'` 别名记录
- `'SOA'` 授权记录的初始值

错误码

每次 DNS 查询都可能返回以下错误码：

- `dns.NODATA`：无数据响应。
- `dns.FORMERR`：查询格式错误。
- `dns.SERVFAIL`：常规失败。
- `dns.NOTFOUND`：没有找到域名。
- `dns.NOTIMP`：未实现请求的操作。
- `dns.REFUSED`：拒绝查询。
- `dns.BADQUERY`：查询格式错误。
- `dns.BADNAME`：域名格式错误。
- `dns.BADFAMILY`：地址协议不支持。
- `dns.BADRESP`：回复格式错误。
- `dns.CONNREFUSED`：无法连接到 DNS 服务器。
- `dns.TIMEOUT`：连接 DNS 服务器超时。
- `dns.EOF`：文件末端。
- `dns.FILE`：读文件错误。
- `dns.NOMEM`：内存溢出。
- `dns.DESTRUCTION`：通道被摧毁。
- `dns.BADSTR`：字符串格式错误。
- `dns.BADFLAGS`：非法标识符。
- `dns.NONAME`：所给主机不是数字。
- `dns.BADHINTS`：非法HINTS标识符。
- `dns.NOTINITIALIZED`：c c-ares 库尚未初始化。
- `dns.LOADIPHLPAPI`：加载 iphlpapi.dll 出错。
- `dns.ADDRGETNETWORKPARAMS`：无法找到 GetNetworkParams 函数。
- `dns.CANCELLED`：取消 DNS 查询。

实例

创建 main.js 文件，代码如下所示：

```
var dns = require('dns');

dns.lookup('www.github.com', function onLookup(err, address, family) {
   console.log('ip 地址:', address);
   dns.reverse(address, function (err, hostnames) {
   if (err) {
      console.log(err.stack);
   }

   console.log('反向解析 ' + address + ': ' + JSON.stringify(hostnames));
});
});
```

执行以上代码，结果如下所示:

```
address: 192.30.252.130
reverse for 192.30.252.130: ["github.com"]
```

Node.js Domain 模块

Node.js **Domain(域)** 简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的异常。引入 Domain 模块 语法格式如下：

```
var domain = require("domain")
```

domain模块，把处理多个不同的IO的操作作为一个组。注册事件和回调到domain，当发生一个错误事件或抛出一个错误时，domain对象会被通知，不会丢失上下文环境，也不导致程序错误立即推出，与process.on('uncaughtException')不同。

Domain 模块可分为隐式绑定和显式绑定：

- 隐式绑定: 把在domain上下文中定义的变量，自动绑定到domain对象
- 显式绑定: 把不是在domain上下文中定义的变量，以代码的方式绑定到domain对象

方法

方法	描述
domain.run(function)	在域的上下文运行提供的函数，隐式的绑定了所有的事件分发器，计时器和底层请求。
domain.add(emitter)	显式的增加事件
domain.remove(emitter)	删除事件。
domain.bind(callback)	返回的函数是一个对于所提供的回调函数的包装函数。当调用这个返回的函数被时，所有被抛出的错误都会被导向到这个域的 error 事件。
domain.intercept(callback)	和 domain.bind(callback) 类似。除了捕捉被抛出的错误外，它还会拦截 Error 对象作为参数传递到这个函数。
domain.enter()	进入一个异步调用的上下文，绑定到domain。
domain.exit()	退出当前的domain，切换到不同的链的异步调用的上下文中。对应domain.enter()。
domain.dispose()	释放一个domain对象，让node进程回收这部分资源。
domain.create()	返回一个domain对象。

属性

属性	描述
domain.members	已加入domain对象的域定时器和事件发射器的数组。

实例

创建 main.js 文件，代码如下所示：

```
var EventEmitter = require("events").EventEmitter;
var domain = require("domain");

var emitter1 = new EventEmitter();

// 创建域
var domain1 = domain.create();

domain1.on('error', function(err){
   console.log("domain1 处理这个错误 ("+err.message+")");
});

// 显式绑定
domain1.add(emitter1);

emitter1.on('error',function(err){
   console.log("监听器处理此错误 ("+err.message+")");
});

emitter1.emit('error',new Error('通过监听器来处理'));

emitter1.removeAllListeners('error');

emitter1.emit('error',new Error('通过 domain1 处理'));

var domain2 = domain.create();

domain2.on('error', function(err){
   console.log("domain2 处理这个错误 ("+err.message+")");
});

// 隐式绑定
domain2.run(function(){
   var emitter2 = new EventEmitter();
   emitter2.emit('error',new Error('通过 domain2 处理'));
});

domain1.remove(emitter1);
emitter1.emit('error', new Error('转换为异常，系统将崩溃!'));
```

执行以上代码，结果如下所示:

```
监听器处理此错误 (通过监听器来处理)
domain1 处理这个错误 (通过 domain1 处理)
domain2 处理这个错误 (通过 domain2 处理)

events.js:72
        throw er; // Unhandled 'error' event
              ^
Error: 转换为异常，系统将崩溃!
    at Object.<anonymous> (/www/node/main.js:40:24)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:929:3
```

Node.js Web 模块

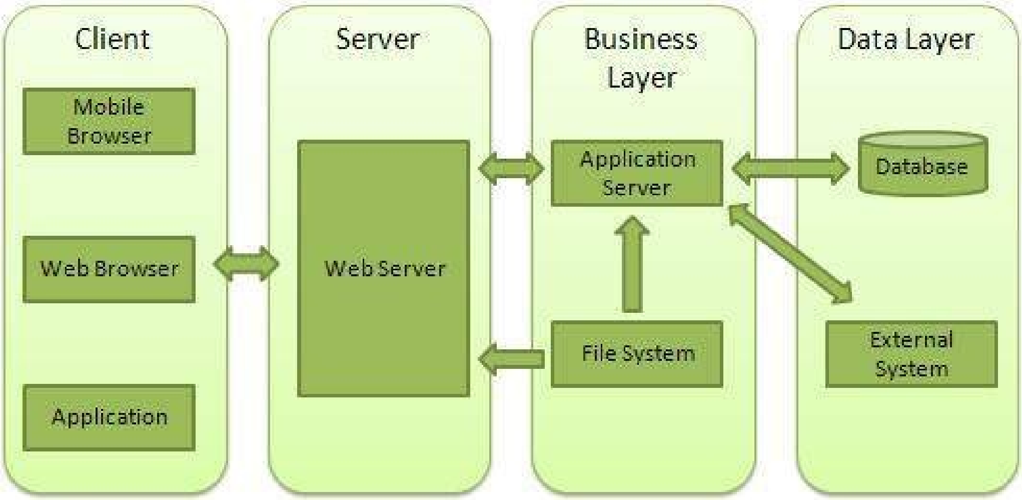
什么是 Web 服务器？

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，Web服务器的基本功能就是提供Web信息浏览服务。它只需支持HTTP协议、HTML文档格式及URL，与客户端的网络浏览器配合。

大多数 web 服务器都支持服务端的脚本语言（php、python、ruby）等，并通过脚本语言从数据库获取数据，将结果返回给客户端浏览器。

目前最主流的三个Web服务器是Apache、Nginx、IIS。

Web 应用架构



- **Client** - 客户端，一般指浏览器，浏览器可以通过 HTTP 协议向服务器请求数据。
- **Server** - 服务端，一般指 Web 服务器，可以接收客户端请求，并向客户端发送响应数据。
- **Business** - 业务层， 通过 Web 服务器处理应用程序，如与数据库交互，逻辑运算，调用外部程序等。
- **Data** - 数据层，一般由数据库组成。

使用 Node 创建 Web 服务器

Node.js 提供了 http 模块，http 模块主要用于搭建 HTTP 服务端和客户端，使用 HTTP 服务器或客户端功能必须调用 http 模块，代码如下：

```
var http = require('http');
```

以下是演示一个最基本的 HTTP 服务器架构(使用8081端口)，创建 server.js 文件，代码如下所示：

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// 创建服务器
http.createServer( function (request, response) {
   // 解析请求，包括文件名
   var pathname = url.parse(request.url).pathname;

   // 输出请求的文件名
   console.log("Request for " + pathname + " received.");

   // 从文件系统中读取请求的文件内容
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);
         // HTTP 状态码: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type': 'text/html'});
      }else{
         // HTTP 状态码: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type': 'text/html'});

         // 响应文件内容
         response.write(data.toString());
      }
      //  发送响应数据
      response.end();
   });
}).listen(8081);

// 控制台会输出以下信息
console.log('Server running at http://127.0.0.1:8081/');
```

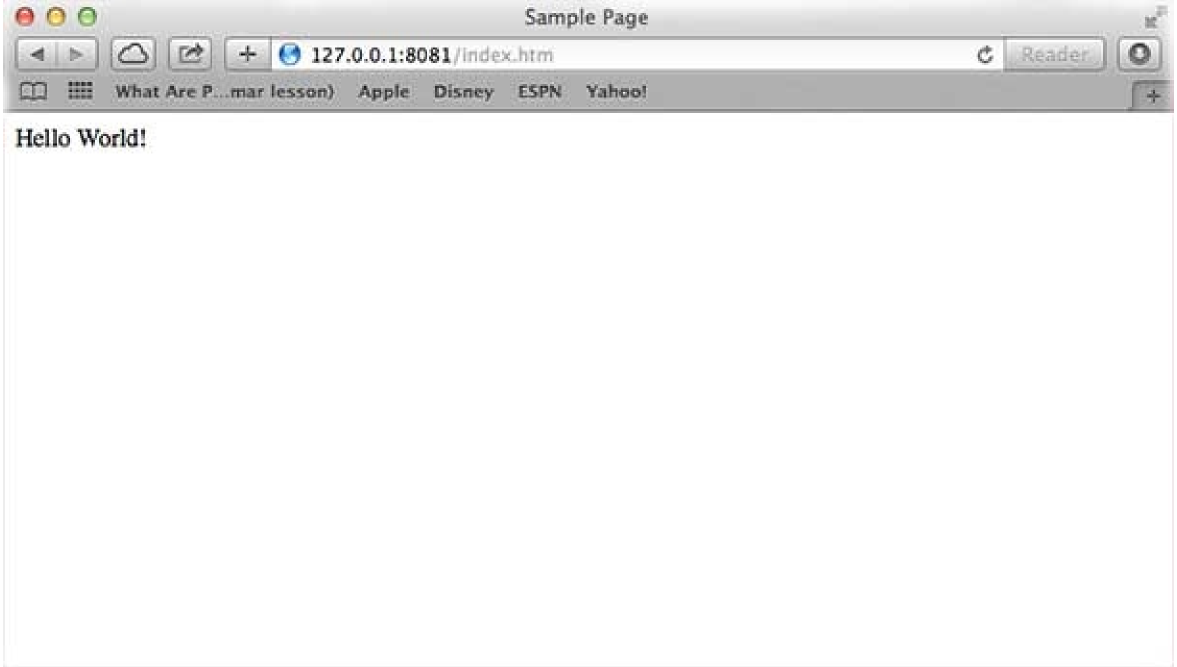
接下来我们在该目录下创建一个 index.htm 文件，代码如下：

```
<html>
<head>
<title>Sample Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

执行 server.js 文件：

```
$ node server.js
Server running at http://127.0.0.1:8081/
```

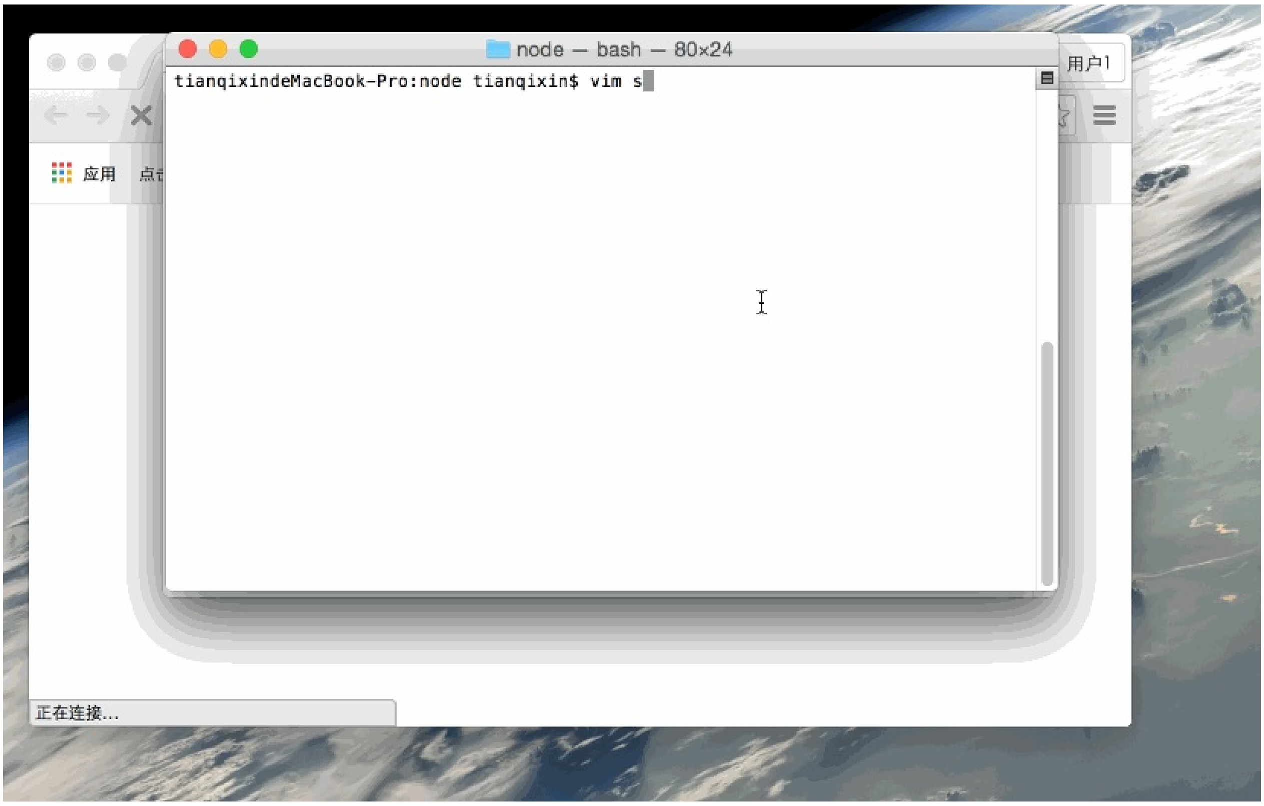
接着我们在浏览器中打开地址：http://127.0.0.1:8081/index.htm，显示如下图所示:



执行 server.js 的控制台输出信息如下：

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.     #  客户端请求信息
```

Gif 实例演示



使用 Node 创建 Web 客户端

Node 创建 Web 客户端需要引入 http 模块，创建 client.js 文件，代码如下所示：

```
<pre>
var http = require('http');

// 用于请求的选项
var options = {
   host: 'localhost',
   port: '8081',
   path: '/index.htm'
};

// 处理响应的回调函数
var callback = function(response){
   // 不断更新数据
   var body = '';
   response.on('data', function(data) {
      body += data;
   });

   response.on('end', function() {
      // 数据接收完成
      console.log(body);
   });
}
// 向服务端发送请求
var req = http.request(options, callback);
req.end();
```

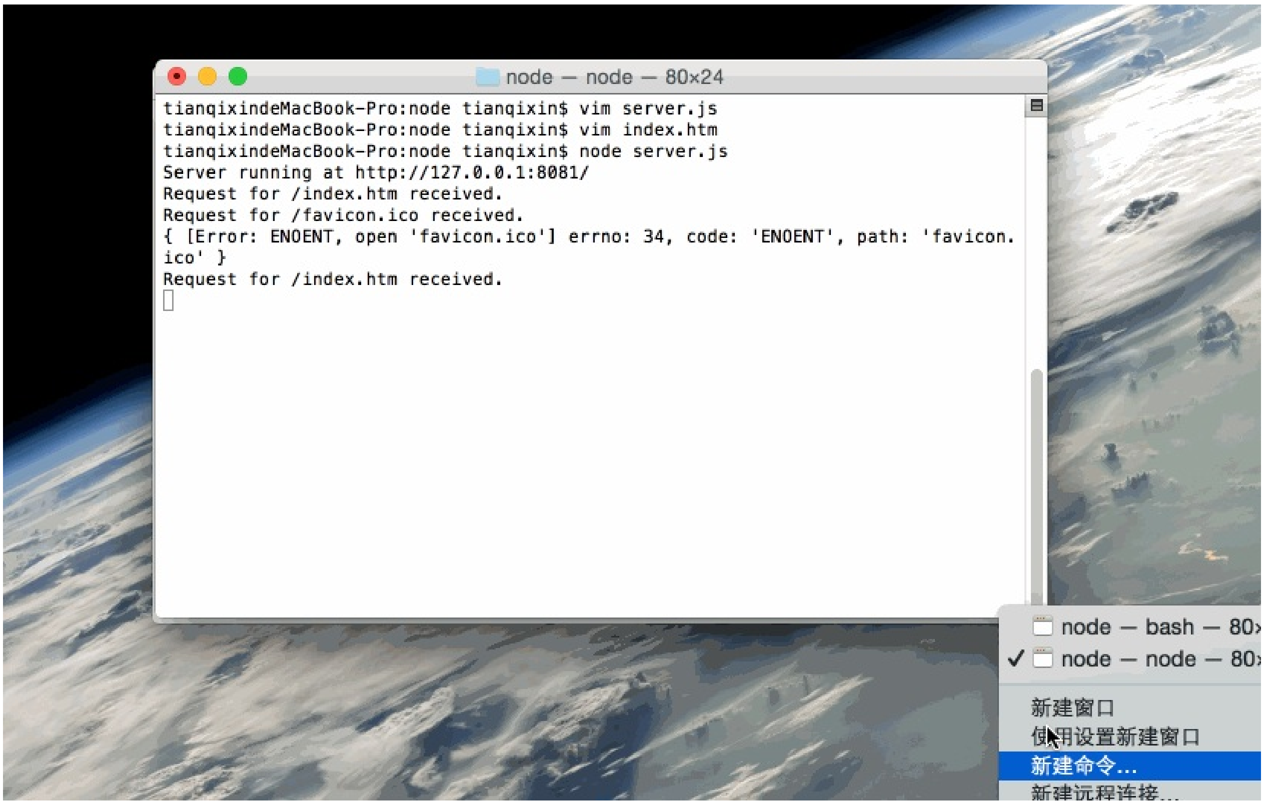
新开一个终端，执行 client.js 文件，输出结果如下：

```
$ node client.js
<html>
<head>
<title>Sample Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

执行 server.js 的控制台输出信息如下：

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.   # 客户端请求信息
```

Gif 实例演示


node — node — 80×24
tianqixindeMacBook-Pro:node tianqixin$ vim server.js
tianqixindeMacBook-Pro:node tianqixin$ vim index.htm
tianqixindeMacBook-Pro:node tianqixin$ node server.js
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
Request for /favicon.ico received.
{ [Error: ENOENT, open 'favicon.ico'] errno: 34, code: 'ENOENT', path: 'favicon.ico' }
Request for /index.htm received.
新建窗口
新建命令...

Node.js Express 框架

Express 简介

Express 是一个简洁而灵活的 node.js Web应用框架, 提供了一系列强大特性帮助你创建各种 Web 应用，和丰富的 HTTP 工具。

使用 Express 可以快速地搭建一个完整功能的网站。

Express 框架核心特性：

- 可以设置中间件来响应 HTTP 请求。
- 定义了路由表用于执行不同的 HTTP 请求动作。
- 可以通过向模板传递参数来动态渲染 HTML 页面。

安装 Express

安装 Express 并将其保存到依赖列表中：

```
$ npm install express --save
```

以上命令会将 Express 框架安装在当期目录的 **node_modules** 目录中， **node_modules** 目录下会自动创建 express 目录。以下几个重要的模块是需要与 express 框架一起安装的：

- **body-parser** - node.js 中间件，用于处理 JSON, Raw, Text 和 URL 编码的数据。
- **cookie-parser** - 这就是一个解析Cookie的工具。通过req.cookies可以取到传过来的cookie，并把它们转成对象。
- **multer** - node.js 中间件，用于处理 enctype="multipart/form-data"（设置表单的MIME编码）的表单数据。

```
$ npm install body-parser --save
$ npm install cookie-parser --save
$ npm install multer --save
```

第一个 Express 框架实例

接下来我们使用 Express 框架来输出 "Hello World"。

以下实例中我们引入了 express 模块，并在客户端发起请求后，响应 "Hello World" 字符串。

创建 express_demo.js 文件，代码如下所示：

```
//express_demo.js 文件
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081，结果如下图所示：

请求和响应

Express 应用使用回调函数的参数： **request** 和 **response** 对象来处理请求和响应的数据。

```
app.get('/', function (req, res) {
   // --
})
```

request 和 **response** 对象的具体介绍：

Request 对象 - request 对象表示 HTTP 请求，包含了请求查询字符串，参数，内容，HTTP 头部等属性。常见属性有：

1. req.app：当callback为外部文件时，用req.app访问express的实例
2. req.baseUrl：获取路由当前安装的URL路径
3. req.body / req.cookies：获得「请求主体」/ Cookies
4. req.fresh / req.stale：判断请求是否还「新鲜」
5. req.hostname / req.ip：获取主机名和IP地址
6. req.originalUrl：获取原始请求URL
7. req.params：获取路由的parameters
8. req.path：获取请求路径
9. req.protocol：获取协议类型
10. req.query：获取URL的查询参数串
11. req.route：获取当前匹配的路由
12. req.subdomains：获取子域名
13. req.accpets（）：检查请求的Accept头的请求类型
14. req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages
15. req.get（）：获取指定的HTTP请求头
16. req.is（）：判断请求头Content-Type的MIME类型

Response 对象 - response 对象表示 HTTP 响应，即在接收到请求时向客户端发送的 HTTP 响应数据。常见属性有：

1. res.app：同req.app一样
2. res.append（）：追加指定HTTP头
3. res.set（）在res.append（）后将重置之前设置的头
4. res.cookie（name，value [，option]）：设置Cookie
5. opition: domain / expires / httpOnly / maxAge / path / secure / signed
6. res.clearCookie（）：清除Cookie
7. res.download（）：传送指定路径的文件
8. res.get（）：返回指定的HTTP头
9. res.json（）：传送JSON响应
10. res.jsonp（）：传送JSONP响应
11. res.location（）：只设置响应的Location HTTP头，不设置状态码或者close response
12. res.redirect（）：设置响应的Location HTTP头，并且设置状态码302
13. res.send（）：传送HTTP响应
14. res.sendFile（path [，options] [，fn]）：传送指定路径的文件 -会自动根据文件

extension设定Content-Type
15. res.set（）：设置HTTP头，传入object可以一次设置多个头
16. res.status（）：设置HTTP状态码
17. res.type（）：设置Content-Type的MIME类型

路由

我们已经了解了 HTTP 请求的基本应用，而路由决定了由谁(指定脚本)去响应客户端请求。

在HTTP请求中，我们可以通过路由提取出请求的URL以及GET/POST参数。

接下来我们扩展 Hello World，添加一些功能来处理更多类型的 HTTP 请求。

创建 express_demo2.js 文件，代码如下所示：

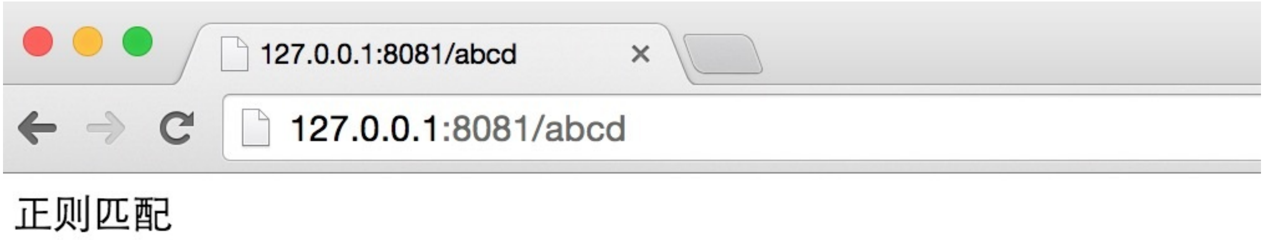
```
var express = require('express');
var app = express();

//  主页输出 "Hello World"
app.get('/', function (req, res) {
   console.log("主页 GET 请求");
   res.send('Hello GET');
})

//  POST 请求
app.post('/', function (req, res) {
   console.log("主页 POST 请求");
   res.send('Hello POST');
})

//  /del_user 页面响应
app.delete('/del_user', function (req, res) {
   console.log("/del_user 响应 DELETE 请求");
   res.send('删除页面');
})

//  /list_user 页面 GET 请求
app.get('/list_user', function (req, res) {
   console.log("/list_user GET 请求");
   res.send('用户列表页面');
})

// 对页面 abcd, abxcd, ab123cd, 等响应 GET 请求
app.get('/ab*cd', function(req, res) {
   console.log("/ab*cd GET 请求");
   res.send('正则匹配');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

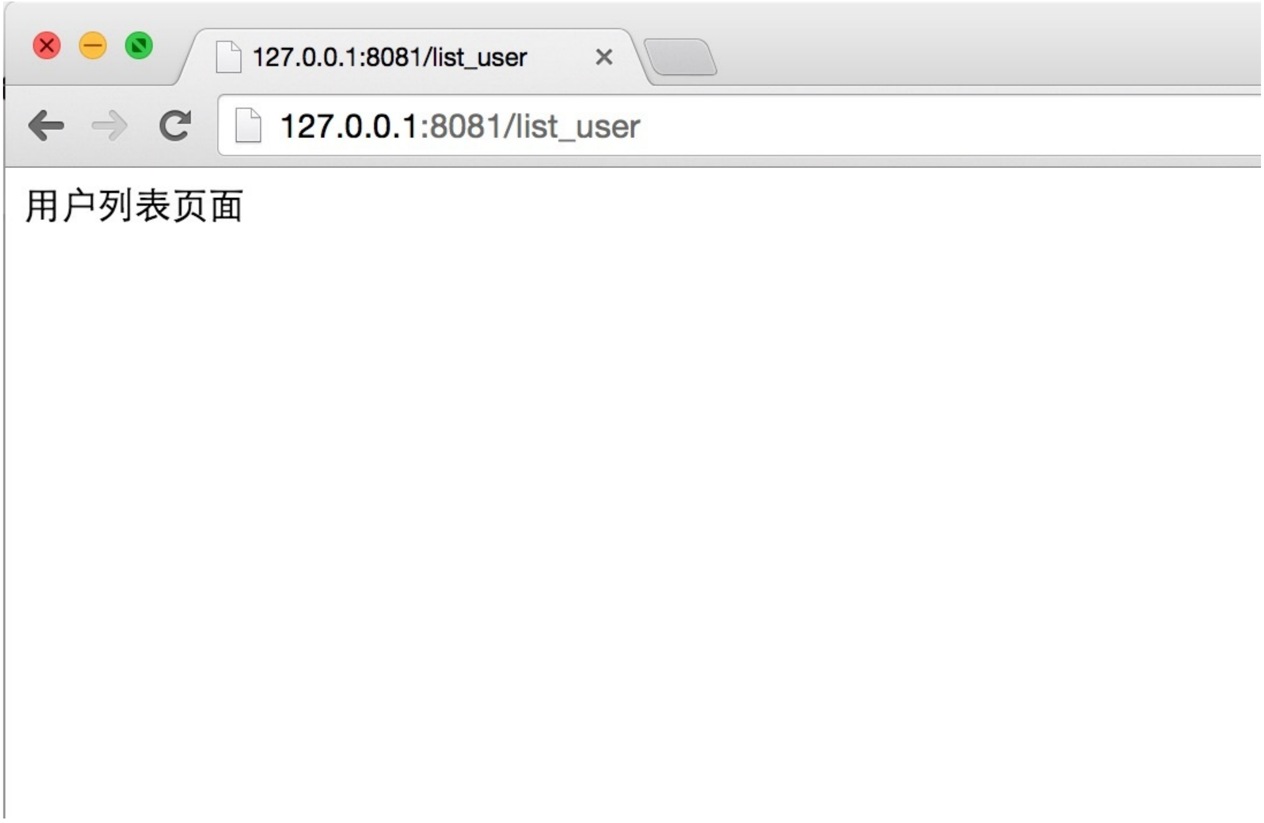
```
$ node express_demo2.js
应用实例，访问地址为 http://0.0.0.0:8081
```

接下来你可以尝试访问 http://127.0.0.1:8081 不同的地址，查看效果。

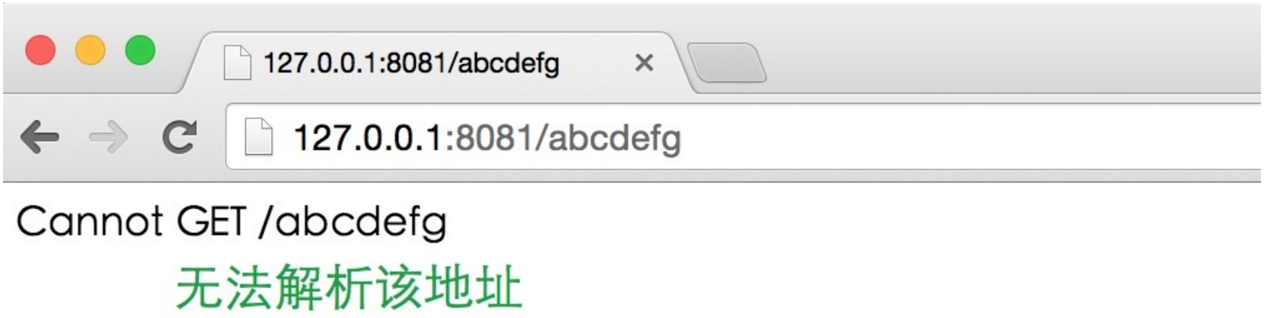
在浏览器中访问 http://127.0.0.1:8081/list_user，结果如下图所示：



在浏览器中访问 http://127.0.0.1:8081/abcd，结果如下图所示：



在浏览器中访问 http://127.0.0.1:8081/abcdefg，结果如下图所示：



静态文件

Express 提供了内置的中间件 **express.static** 来设置静态文件如：图片， CSS, JavaScript 等。

你可以使用 **express.static** 中间件来设置静态文件路径。例如，如果你将图片， CSS, JavaScript 文件放在 public 目录下，你可以这么写：

```
app.use(express.static('public'));
```

我们可以到 public/images 目录下放些图片,如下所示：

```
node_modules
server.js
public/
public/images
public/images/logo.png
```

让我们再修改下 "Hello Word" 应用添加处理静态文件的功能。

创建 express_demo3.js 文件，代码如下所示：

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo3.js
应用实例，访问地址为 http://0.0.0.0:8081
```

执行以上代码：

在浏览器中访问 http://127.0.0.1:8081/images/logo.png（本实例采用了菜鸟教程的logo），结果如下图所示：

RUNOOB.COM

GET 方法

以下实例演示了在表单中通过 GET 方法提交两个参数，我们可以使用 server.js 文件内的 **process_get** 路由器来处理输入：

index.htm 文件代码如下：

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_get" method="GET">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

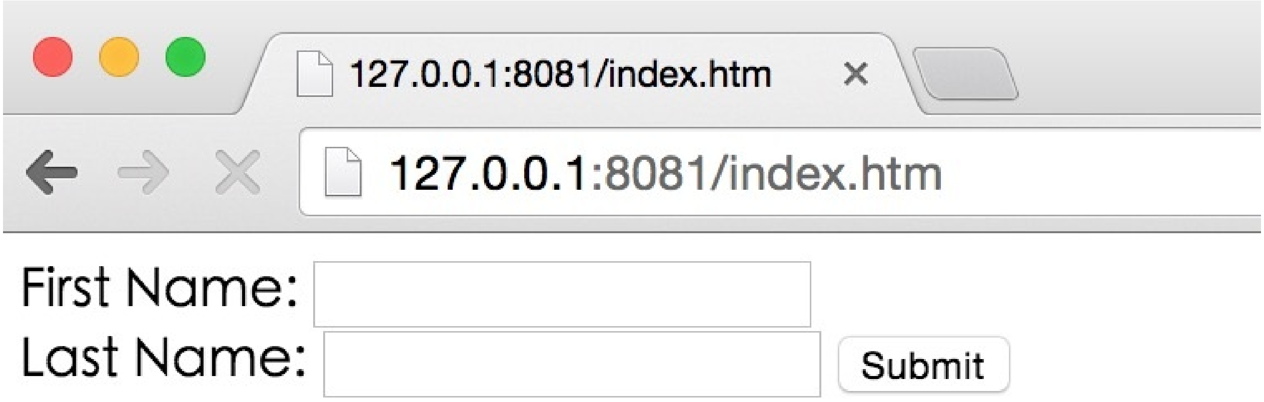
server.js 文件代码如下:

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.get('/process_get', function (req, res) {

   // 输出 JSON 格式
   response = {
       first_name:req.query.first_name,
       last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

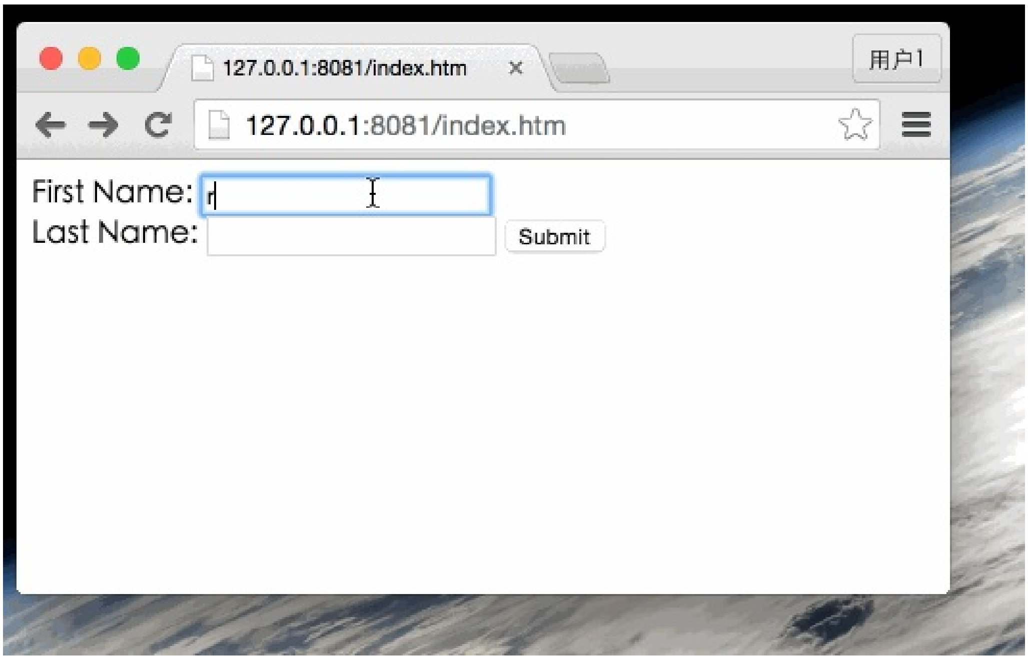
执行以上代码：

```
node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



POST 方法

以下实例演示了在表单中通过 POST 方法提交两个参数，我们可以使用 server.js 文件内的 **process_get** 路由器来处理输入：

index.htm 文件代码修改如下：

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_post" method="POST">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

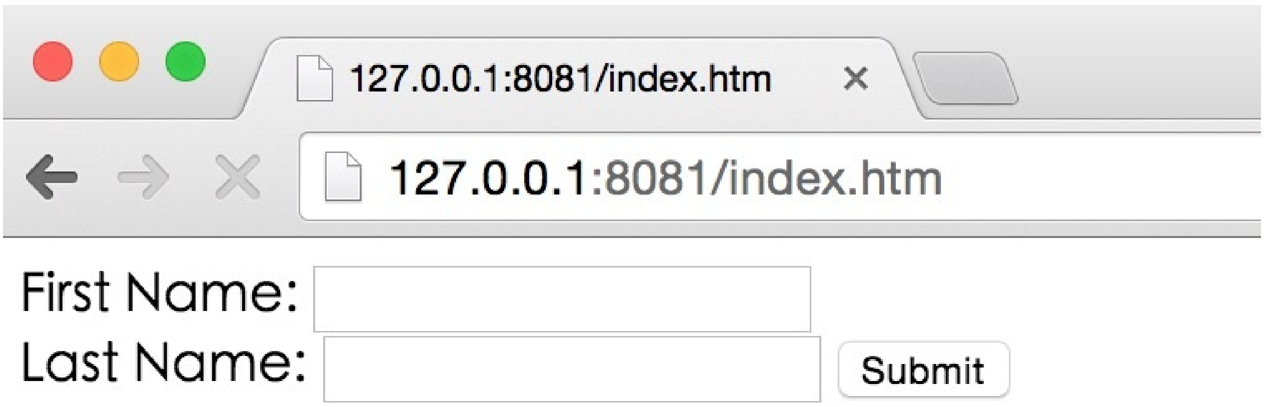
server.js 文件代码修改如下:

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

// 创建 application/x-www-form-urlencoded 编码解析
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/process_post', urlencodedParser, function (req, res) {

   // 输出 JSON 格式
   response = {
       first_name:req.body.first_name,
       last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

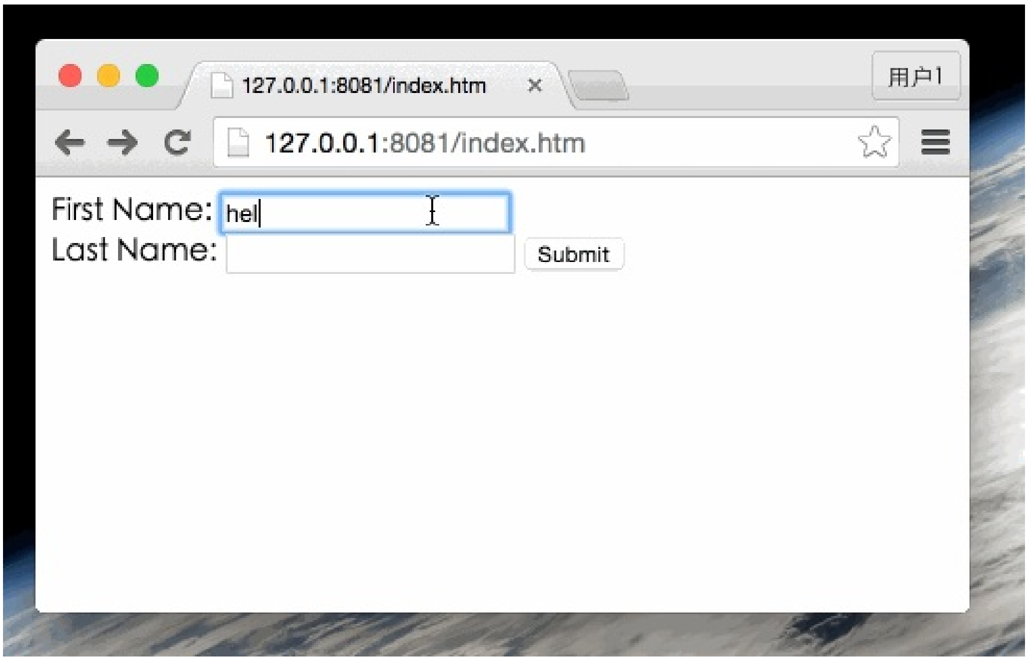
执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



文件上传

以下我们创建一个用于上传文件的表单，使用 POST 方法，表单 enctype 属性设置为 multipart/form-data。

index.htm 文件代码修改如下：

```
<html>
<head>
<title>文件上传表单</title>
</head>
<body>
<h3>文件上传：</h3>
选择一个文件上传: <br />
<form action="/file_upload" method="post" enctype="multipart/form-data">
<input type="file" name="image" size="50" />
<br />
<input type="submit" value="上传文件" />
</form>
</body>
</html>
```

server.js 文件代码修改如下:

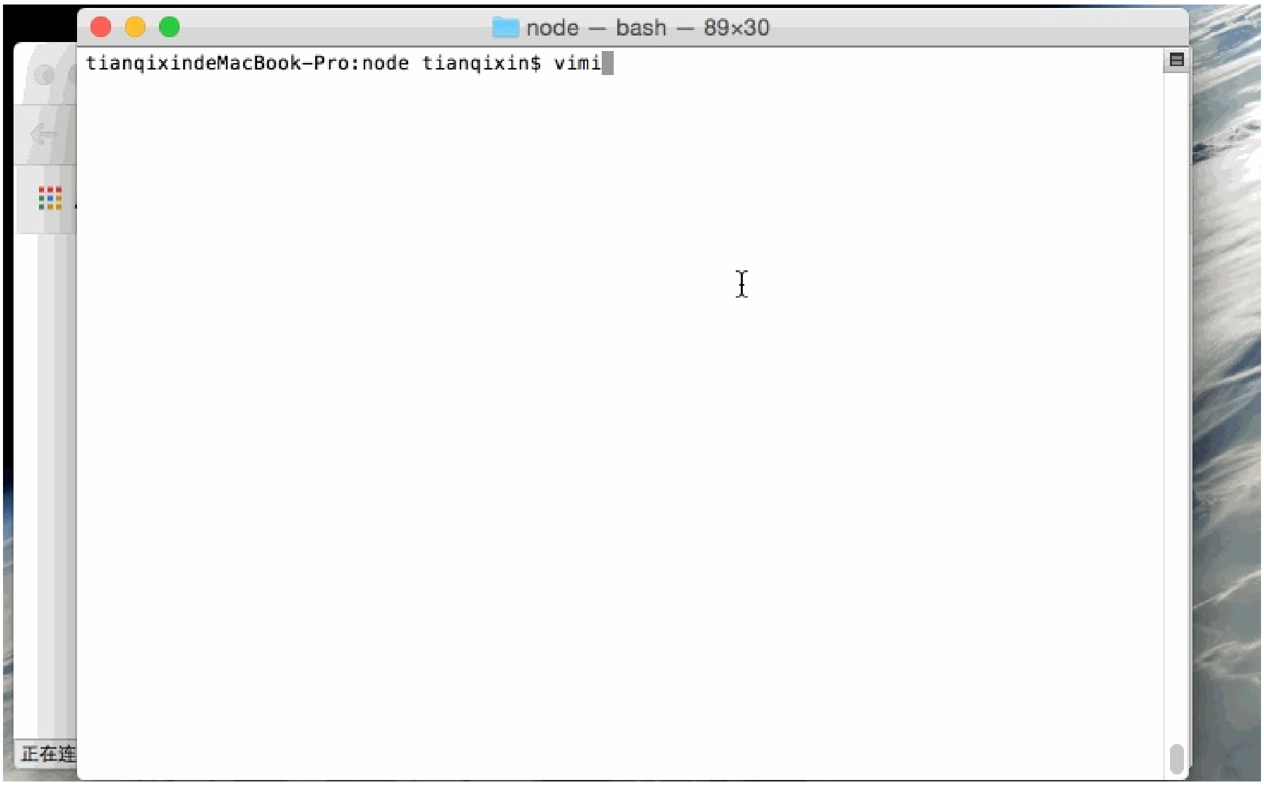
```
var express = require('express');
var app = express();
var fs = require("fs");

var bodyParser = require('body-parser');
var multer  = require('multer');

app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/'}).array('image'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/file_upload', function (req, res) {

   console.log(req.files[0]);  // 上传的文件信息

   var des_file = __dirname + "/" + req.files[0].originalname;
   fs.readFile( req.files[0].path, function (err, data) {
        fs.writeFile(des_file, data, function (err) {
         if( err ){
              console.log( err );
         }else{
               response = {
                   message:'File uploaded successfully',
                   filename:req.files[0].originalname
              };
          }
          console.log( response );
          res.end( JSON.stringify( response ) );
       });
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：

现在你可以向表单输入数据，并提交，如下演示：



Cookie 管理

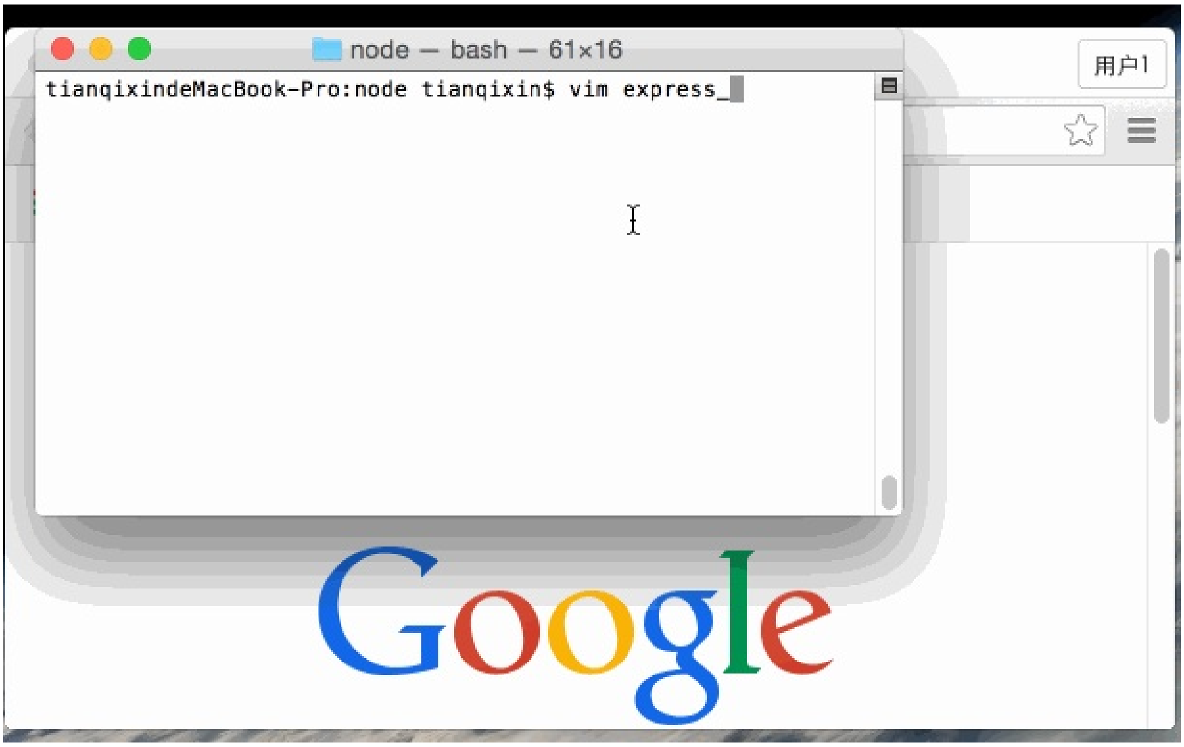
我们可以使用中间件向 Node.js 服务器发送 cookie 信息，以下代码输出了客户端发送的 cookie 信息：

```
// express_cookie.js 文件
var express      = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser())

app.get('/', function(req, res) {
  console.log("Cookies: ", req.cookies)
})

app.listen(8081)
```

执行以上代码：

```
$ node express_demo.js
```

现在你可以访问 http://127.0.0.1:8081 并查看终端信息的输出，如下演示：



Node.js RESTful API

什么是 REST？

RREST即表述性状态传递（英文：Representational State Transfer，简称REST）是Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格。

表述性状态转移是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful。需要注意的是，REST是设计风格而不是标准。REST通常基于使用HTTP，URI，和XML（标准通用标记语言下的一个子集）以及HTML（标准通用标记语言下的一个应用）这些现有的广泛流行的协议和标准。REST 通常使用 JSON 数据格式。

HTTP 方法

以下为 REST 基本架构的四个方法：

- **GET** - 用于获取数据。
- **PUT** - 用于添加数据。
- **DELETE** - 用于删除数据。
- **POST** - 用于更新或添加数据。

RESTful Web Services

Web service是一个平台独立的，低耦合的，自包含的、基于可编程的web的应用程序，可使用开放的XML（标准通用标记语言下的一个子集）标准来描述、发布、发现、协调和配置这些应用程序，用于开发分布式的互操作的应用程序。

基于 REST 架构的 Web Services 即是 RESTful。

由于轻量级以及通过 HTTP 直接传输数据的特性，Web 服务的 RESTful 方法已经成为最常见的替代方法。可以使用各种语言（比如 Java 程序、Perl、Ruby、Python、PHP 和 Javascript[包括 Ajax]）实现客户端。

RESTful Web 服务通常可以通过自动客户端或代表用户的应用程序访问。但是，这种服务的简便性让用户能够与之直接交互，使用它们的 Web 浏览器构建一个 GET URL 并读取返回的内容。

更多介绍，可以查看：RESTful 架构详解

创建 RESTful

首先，创建一个 json 数据资源文件 users.json，内容如下：

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },
   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },
   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

基于以上数据，我们创建以下 RESTful API：

序号	**URI**	**HTTP 方法**	发送内容	结果
1	listUsers	GET	空	显示所有用户列表
2	addUser	POST	JSON 字符串	添加新用户
3	deleteUser	DELETE	JSON 字符串	删除用户
4	:id	GET	空	显示用户详细信息

获取用户列表：

以下代码，我们创建了 RESTful API **listUsers**，用于读取用户的信息列表， server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/listUsers', function (req, res) {
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       console.log( data );
       res.end( data );
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/listUsers，结果如下所示：

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },
   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },
   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

添加用户

以下代码，我们创建了 RESTful API **addUser**， 用于添加新的用户数据，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

//添加的新用户数据
var user = {
   "user4" : {
      "name" : "mohit",
      "password" : "password4",
      "profession" : "teacher",
      "id": 4
   }
}

app.get('/addUser', function (req, res) {
   // 读取已存在的数据
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       data["user4"] = user["user4"];
       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/addUsers，结果如下所示：

```
{ user1:
   { name: 'mahesh',
     password: 'password1',
     profession: 'teacher',
     id: 1 },
  user2:
   { name: 'suresh',
     password: 'password2',
     profession: 'librarian',
     id: 2 },
  user3:
   { name: 'ramesh',
     password: 'password3',
     profession: 'clerk',
     id: 3 },
  user4:
   { name: 'mohit',
     password: 'password4',
     profession: 'teacher',
     id: 4 }
}
```

显示用户详情

以下代码，我们创建了 RESTful API **:id**（用户**id**）， 用于读取指定用户的详细信息，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/:id', function (req, res) {
   // 首先我们读取已存在的用户
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       var user = data["user" + req.params.id]
       console.log( user );
       res.end( JSON.stringify(user));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/2，结果如下所示：

```
{
   "name":"suresh",
   "password":"password2",
   "profession":"librarian",
   "id":2
}
```

删除用户

以下代码，我们创建了 RESTful API **deleteUser**， 用于删除指定用户的详细信息，以下实例中，用户 id 为 2，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

var id = 2;

app.get('/deleteUser', function (req, res) {

   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       delete data["user" + 2];

       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/deleteUser，结果如下所示：

```
{ user1:
   { name: 'mahesh',
     password: 'password1',
     profession: 'teacher',
     id: 1 },
  user3:
   { name: 'ramesh',
     password: 'password3',
     profession: 'clerk',
     id: 3 }
}
```

Node.js 多进程

我们都知道 Node.js 是以单线程的模式运行的，但它使用的是事件驱动来处理并发，这样有助于我们在多核 cpu 的系统上创建多个子进程，从而提高性能。

每个子进程总是带有三个流对象：child.stdin, child.stdout 和child.stderr。他们可能会共享父进程的 stdio 流，或者也可以是独立的被导流的流对象。

Node 提供了 child_process 模块来创建子进程，方法有：

- **exec** - child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。
- **spawn** - child_process.spawn 使用指定的命令行参数创建新线程。
- **fork** - child_process.fork 是 spawn()的特殊形式，用于在子进程中运行的模块，如 fork('./son.js') 相当于 spawn('node', ['./son.js']) 。与spawn方法不同的是，fork会在父进程与子进程之间，建立一个通信管道，用于进程之间的通信。

exec() 方法

child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。

语法如下所示：

```
child_process.exec(command[, options], callback)
```

参数

参数说明如下：

command： 字符串， 将要运行的命令，参数使用空格隔开

options ：对象，可以是：

- cwd ，字符串，子进程的当前工作目录
- env，对象 环境变量键值对
- encoding ，字符串，字符编码（默认： 'utf8'）
- shell ，字符串，将要执行命令的 Shell（默认: 在 UNIX 中为 `/bin/sh` ， 在 Windows 中为 `cmd.exe` ， Shell 应当能识别 `-c` 开关在 UNIX 中，或 `/s /c` 在 Windows 中。 在 Windows 中，命令行解析应当能兼容 `cmd.exe` ）

- timeout，数字，超时时间（默认： 0）
- maxBuffer，数字， 在 stdout 或 stderr 中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死 （默认: 200*1024）
- killSignal ，字符串，结束信号（默认：'SIGTERM'）
- uid，数字，设置用户进程的 ID
- gid，数字，设置进程组的 ID

callback ：回调函数，包含三个参数error, stdout 和 stderr。

exec() 方法返回最大的缓冲区，并等待进程结束，一次性返回缓冲区的内容。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.exec('node support.js '+i,
      function (error, stdout, stderr) {
         if (error) {
            console.log(error.stack);
            console.log('Error code: '+error.code);
            console.log('Signal received: '+error.signal);
         }
         console.log('stdout: ' + stdout);
         console.log('stderr: ' + stderr);
      });

      workerProcess.on('exit', function (code) {
      console.log('子进程已退出，退出码 '+code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js
子进程已退出，退出码 0
stdout: 进程 1 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 0 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 2 执行。

stderr:
```

spawn() 方法

child_process.spawn 使用指定的命令行参数创建新线程，语法格式如下：

```
child_process.spawn(command[, args][, options])
```

参数

参数说明如下：

command： 将要运行的命令

args： Array 字符串参数数组

options Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- stdio Array|String 子进程的 stdio 配置
- detached Boolean 这个子进程将会变成进程组的领导
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

spawn() 方法返回流 (stdout & stderr)，在进程返回大量数据时使用。进程一旦开始执行时 spawn() 就开始接收响应。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.spawn('node', ['support.js', i]);

   workerProcess.stdout.on('data', function (data) {
      console.log('stdout: ' + data);
   });

   workerProcess.stderr.on('data', function (data) {
      console.log('stderr: ' + data);
   });

   workerProcess.on('close', function (code) {
      console.log('子进程已退出，退出码 '+code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js stdout: 进程 0 执行。

子进程已退出，退出码 0
stdout: 进程 1 执行。

子进程已退出，退出码 0
stdout: 进程 2 执行。

子进程已退出，退出码 0
```

fork 方法

child_process.fork 是 spawn() 方法的特殊形式，用于创建进程，语法格式如下：

```
child_process.fork(modulePath[, args][, options])
```

参数

参数说明如下：

modulePath： String，将要在子进程中运行的模块

args： Array 字符串参数数组

options：Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- execPath String 创建子进程的可执行文件
- execArgv Array 子进程的可执行文件的字符串参数数组（默认： process.execArgv）

- silent Boolean 如果为 `true`，子进程的 `stdin`，`stdout` 和 `stderr` 将会被关联至父进程，否则，它们将会从父进程中继承。（默认为：`false`）
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

返回的对象除了拥有ChildProcess实例的所有方法，还有一个内建的通信信道。

h3>实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var worker_process = child_process.fork("support.js", [i]);

   worker_process.on('close', function (code) {
      console.log('子进程已退出，退出码 ' + code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js
进程 0 执行。
子进程已退出，退出码 0
进程 1 执行。
子进程已退出，退出码 0
进程 2 执行。
子进程已退出，退出码 0
```

Node.js JXcore 打包

Node.js 是一个开放源代码、跨平台的、用于服务器端和网络应用的运行环境。

JXcore 是一个支持多线程的 Node.js 发行版本，基本不需要对你现有的代码做任何改动就可以直接线程安全地以多线程运行。

但我们这篇文章主要是要教大家介绍 JXcore 的打包功能。

JXcore 安装

下载 JXcore 安装包，并解压，在解压的的目录下提供了 jx 二进制文件命令，接下来我们主要使用这个命令。

步骤1、下载

下载 JXcore 安装包 http://jxcore.com/downloads/，你需要根据你自己的系统环境来下载安装包。

1、Window 平台下载：Download，

2、Linux/OSX 下载安装命令，直接下载解压包下的 jx 二进制文件拷贝到 /usr/bin 目录下：

```
$ wget https://s3.amazonaws.com/nodejx/jx_rh64.zip
$ unzip jx_rh64.zip
$ cp jx_rh64/jx /usr/bin
```

将 /usr/bin 添加到 PATH 路径中：

```
$ export PATH=$PATH:/usr/bin
```

以上步骤如果操作正确，使用以下命令，会输出版本号信息：

```
$ jx --version
v0.10.32
```

包代码

例如，我们的 Node.js 项目包含以下几个文件，其中 index.js 是主文件：

```
drwxr-xr-x  2 root root  4096 Nov 13 12:42 images
-rwxr-xr-x  1 root root 30457 Mar  6 12:19 index.htm
-rwxr-xr-x  1 root root 30452 Mar  1 12:54 index.js
drwxr-xr-x 23 root root  4096 Jan 15 03:48 node_modules
drwxr-xr-x  2 root root  4096 Mar 21 06:10 scripts
drwxr-xr-x  2 root root  4096 Feb 15 11:56 style
```

接下来我们使用 **jx** 命令打包以上项目，并指定 index.js 为 Node.js 项目的主文件：

```
$ jx package index.js index
```

以上命令执行成功，会生成以下两个文件：

- **index.jxp** 这是一个中间件文件，包含了需要编译的完整项目信息。
- **index.jx** 这是一个完整包信息的二进制文件，可运行在客户端上。

载入 JX 文件

我们使用 **jx** 命令打包项目：

```
$ node index.js command_line_arguments
```

使用 JXcore 编译后，我们可以使用以下命令来执行生成的 jx 二进制文件：

```
$ jx index.jx command_line_arguments
```

更多 JXcore 功能特性你可以参考官网：http://jxcore.com/。

W3School Linux 教程

来源：Linux 教程

整理：飞龙

Linux 基础

Linux 简介

Linux内核最初只是由芬兰人李纳斯·托瓦兹（Linus Torvalds）在赫尔辛基大学上学时出于个人爱好而编写的。

Linux是一套免费使用和自由传播的类Unix操作系统，是一个基于POSIX和UNIX的多用户、多任务、支持多线程和多CPU的操作系统。

Linux能运行主要的UNIX工具软件、应用程序和网络协议。它支持32位和64位硬件。Linux继承了Unix以网络为核心的设计思想，是一个性能稳定的多用户网络操作系统。

Linux的发行版

Linux的发行版说简单点就是将Linux内核与应用软件做一个打包。

目前市面上较知名的发行版有：Ubuntu、RedHat、CentOS、Debain、Fedora、SuSE、OpenSUSE、TurboLinux、BluePoint、RedFlag、Xterm、SlackWare等。

Linux应用领域

今天各种场合都有使用各种Linux发行版，从嵌入式设备到超级计算机，并且在服务器领域确定了地位，通常服务器使用LAMP（Linux + Apache + MySQL + PHP）或LNMP（Linux + Nginx+ MySQL + PHP）组合。

目前Linux不仅在家庭与企业中使用，并且在政府中也很受欢迎。

- 巴西联邦政府由于支持Linux而世界闻名。
- 有新闻报道俄罗斯军队自己制造的Linux发布版的，做为G.H.ost项目已经取得成果.
- 印度的Kerala联邦计划在向全联邦的高中推广使用Linux。
- 中华人民共和国为取得技术独立，在龙芯过程中排他性地使用Linux。
- 在西班牙的一些地区开发了自己的Linux发布版，并且在政府与教育领域广泛使用，如Extremadura地区的gnuLinEx和Andalusia地区的Guadalinex。
- 葡萄牙同样使用自己的Linux发布版Caixa Mágica，用于Magalh?es笔记本电脑和e-escola政府软件。
- 法国和德国同样开始逐步采用Linux。

Linux vs Window

目前国内Linux更多的是应用于服务器上，而桌面操作系统更多使用的是Window。主要区别如下：

比较	Windows	Linux
界面	界面统一，外壳程序固定所有Windows程序菜单几乎一致，快捷键也几乎相同	图形界面风格依发布版不同而不同，可能互不兼容。GNU/Linux的终端机是从UNIX传承下来，基本命令和操作方法也几乎一致。
驱动程序	驱动程序丰富，版本更新频繁。默认安装程序里面一般包含有该版本发布时流行的硬件驱动程序，之后所出的新硬件驱动依赖于硬件厂商提供。对于一些老硬件，如果没有了原配的驱动有时很难支持。另外，有时硬件厂商未提供所需版本的Windows下的驱动，也会比较头痛。	由志愿者开发，由Linux核心开发小组发布，很多硬件厂商基于版权考虑并未提供驱动程序，尽管多数无需手动安装，但是涉及安装则相对复杂，使得新用户面对驱动程序问题（是否存在和安装方法）会一筹莫展。但是在开源开发模式下，许多老硬件尽管在Windows下很难支持的也容易找到驱动。HP、Intel、AMD等硬件厂商逐步不同程度支持开源驱动，问题正在得到缓解。
使用	使用比较简单，容易入门。图形化界面对没有计算机背景知识的用户使用十分有利。	图形界面使用简单，容易入门。文字界面，需要学习才能掌握。
学习	系统构造复杂、变化频繁，且知识、技能淘汰快，深入学习困难。	系统构造简单、稳定，且知识、技能传承性好，深入学习相对容易。
软件	每一种特定功能可能都需要商业软件的支持，需要购买相应的授权。	大部分软件都可以自由获取，同样功能的软件选择较少。

Linux 安装

本章节我们将为大家介绍Linux的安装。

本章节以 centos6.4 为例。

centos6.4 下载地址：

- 网易镜像：http://mirrors.163.com/centos/6/isos/
- 搜狐镜像：http://mirrors.sohu.com/centos/6/isos/

注：建议安装64位Linux系统。

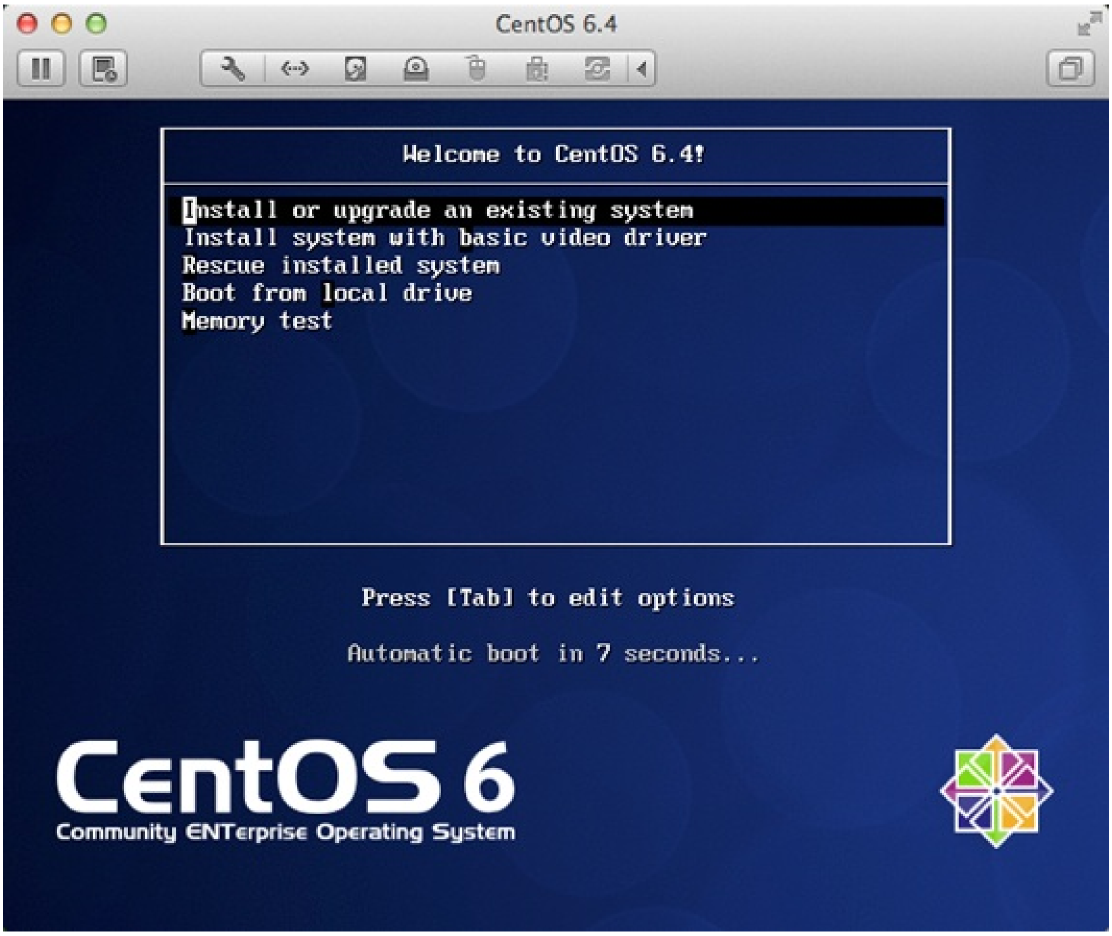
接下来你需要将下载的Linux系统刻录成光盘或U盘。

注：你也可以在Window上安装VMware虚拟机来安装Linux系统。

Linux 安装步骤

1、首先，使用光驱或U盘或你下载的Linux ISO文件进行安装。

界面说明：



Install or upgrade an existing system 安装或升级现有的系统

install system with basic video driver 安装过程中采用基本的显卡驱动

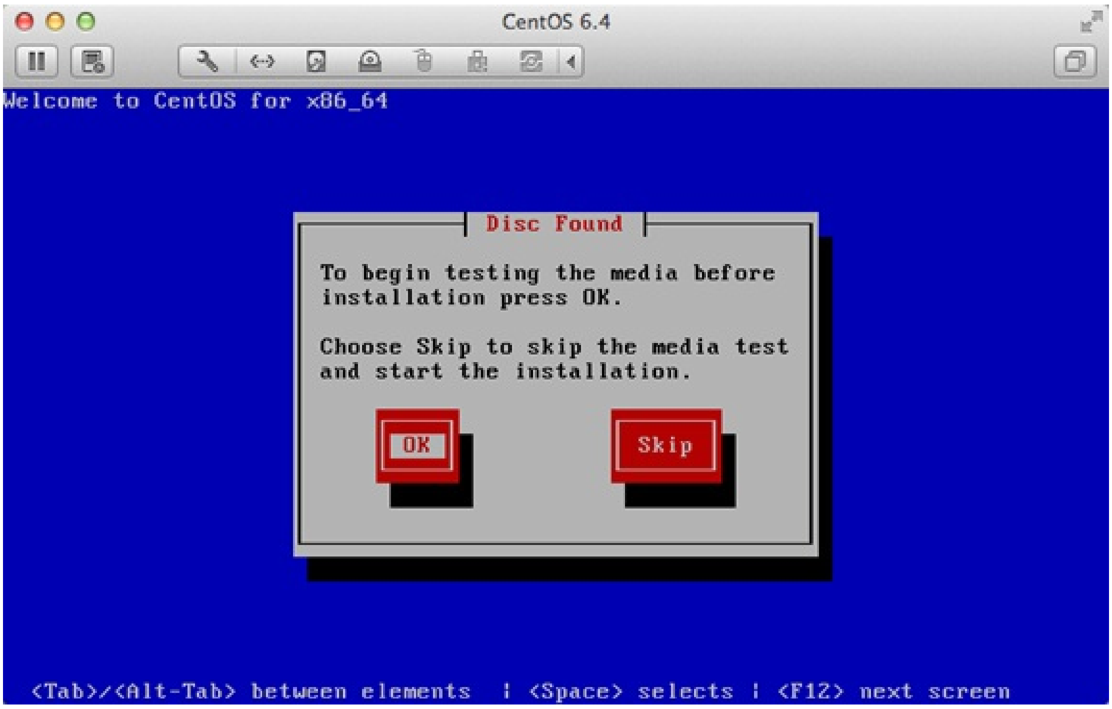
Rescue installed system 进入系统修复模式

Boot from local drive 退出安装从硬盘启动

Memory test 内存检测

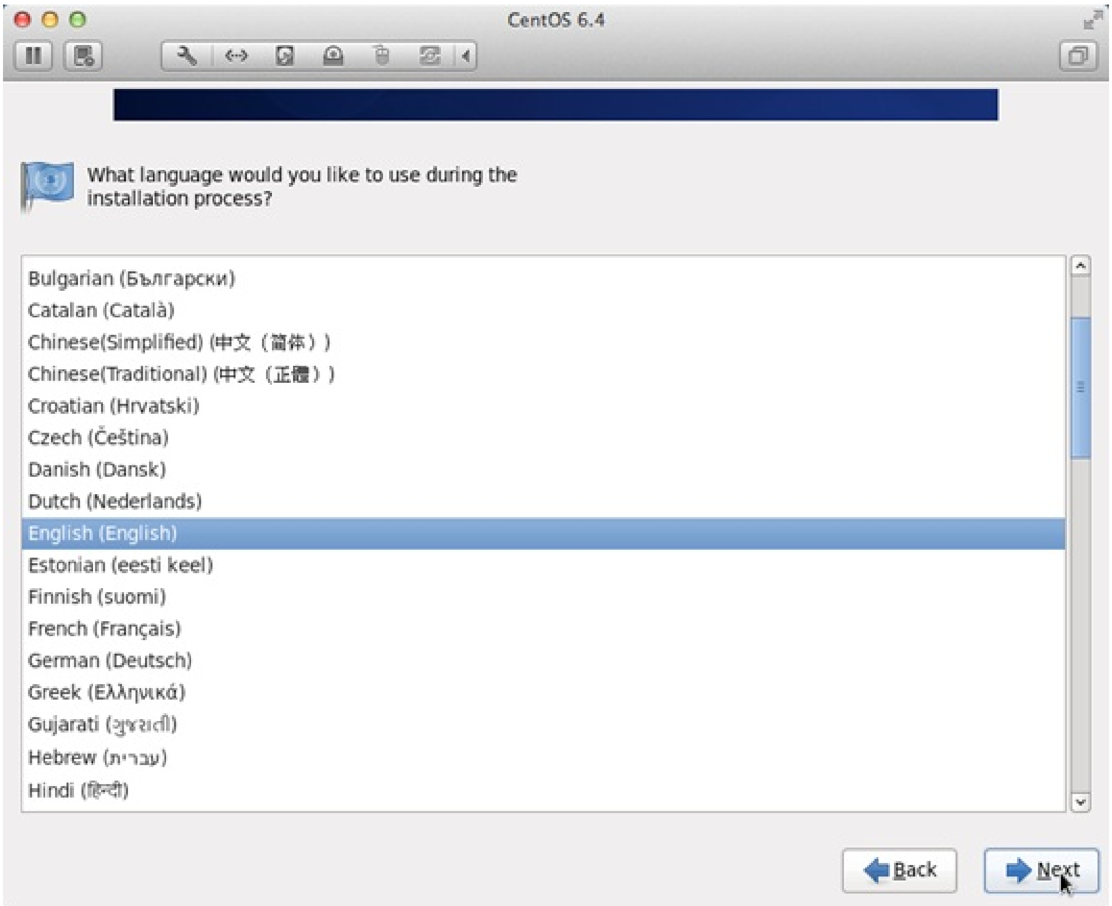
注：用联想E49安装时选择第一项安装时会出现屏幕显示异常的问题，后改用第二项安装时就没有出现问题

2、介质直接"skip"就可以了

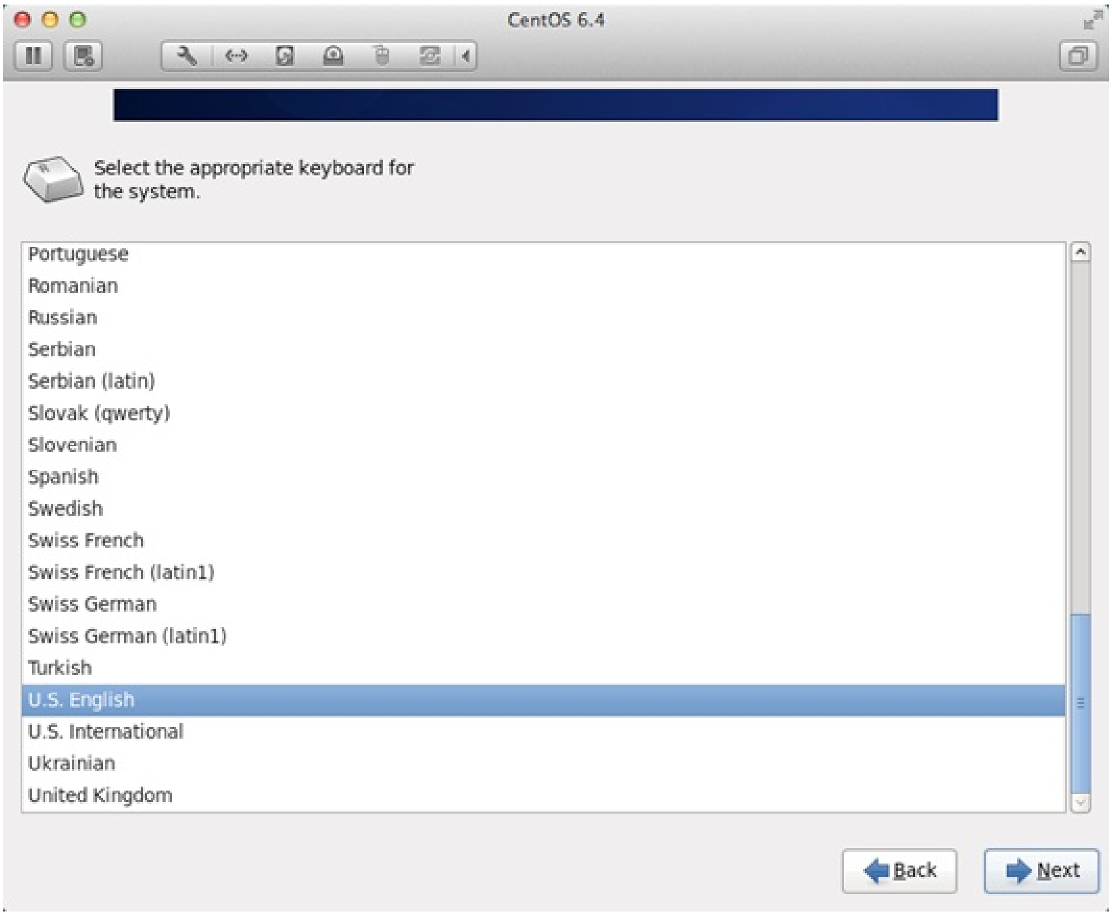


3、出现引导界面，点击"next"

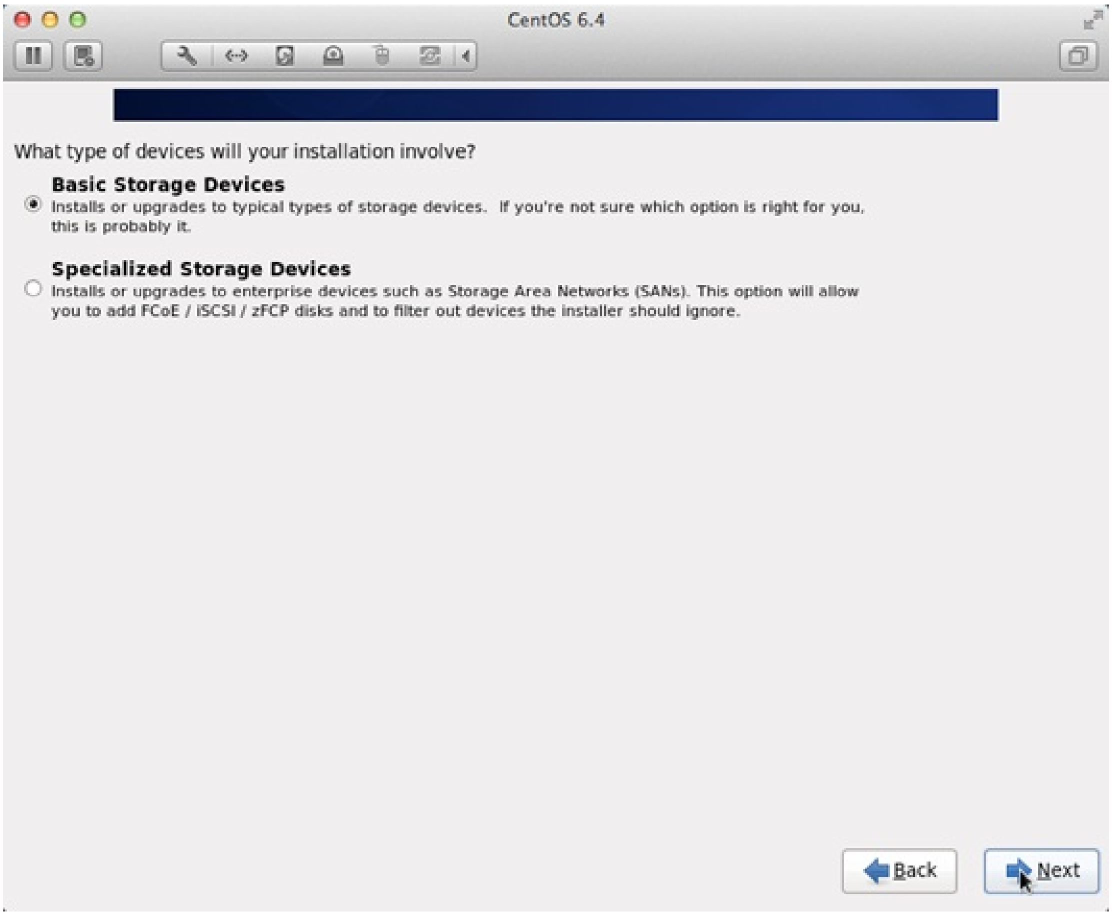
4、选中"English（English）"否则会有部分乱码问题



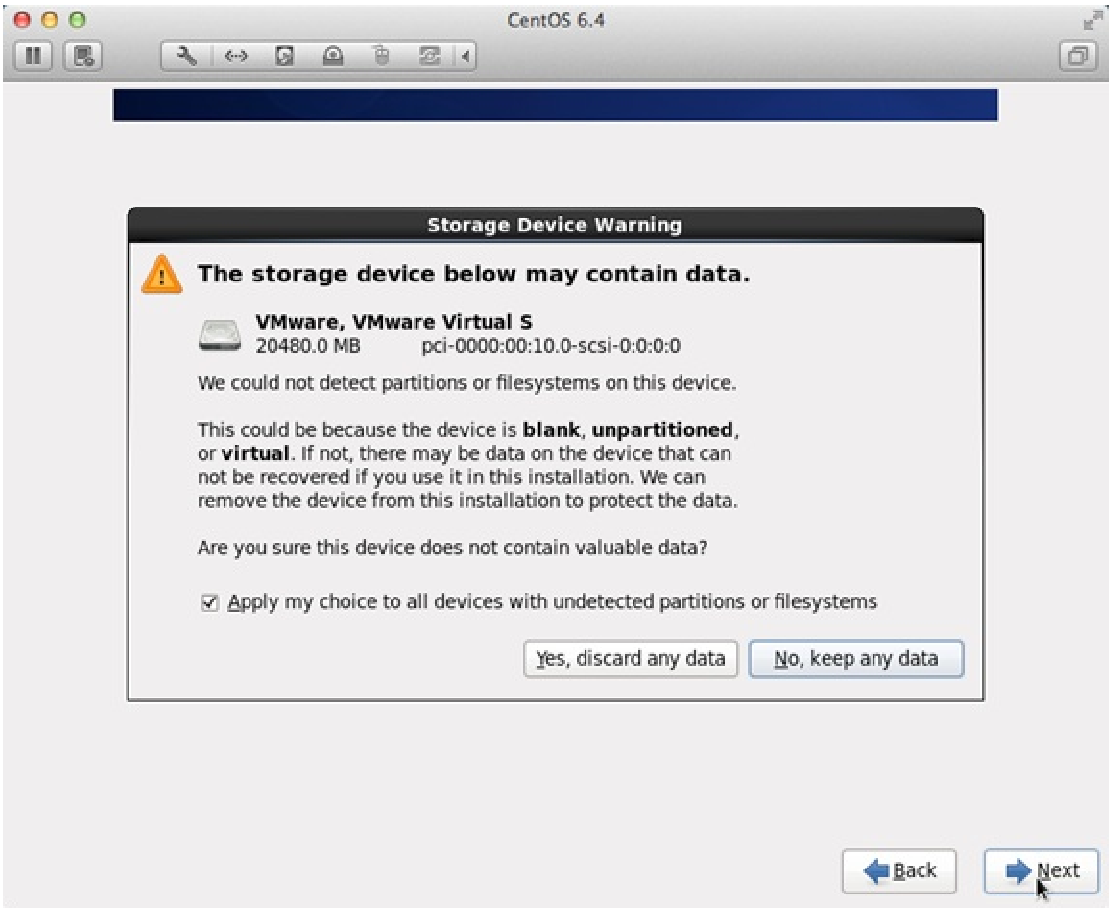
5、键盘布局选择"U.S.English"



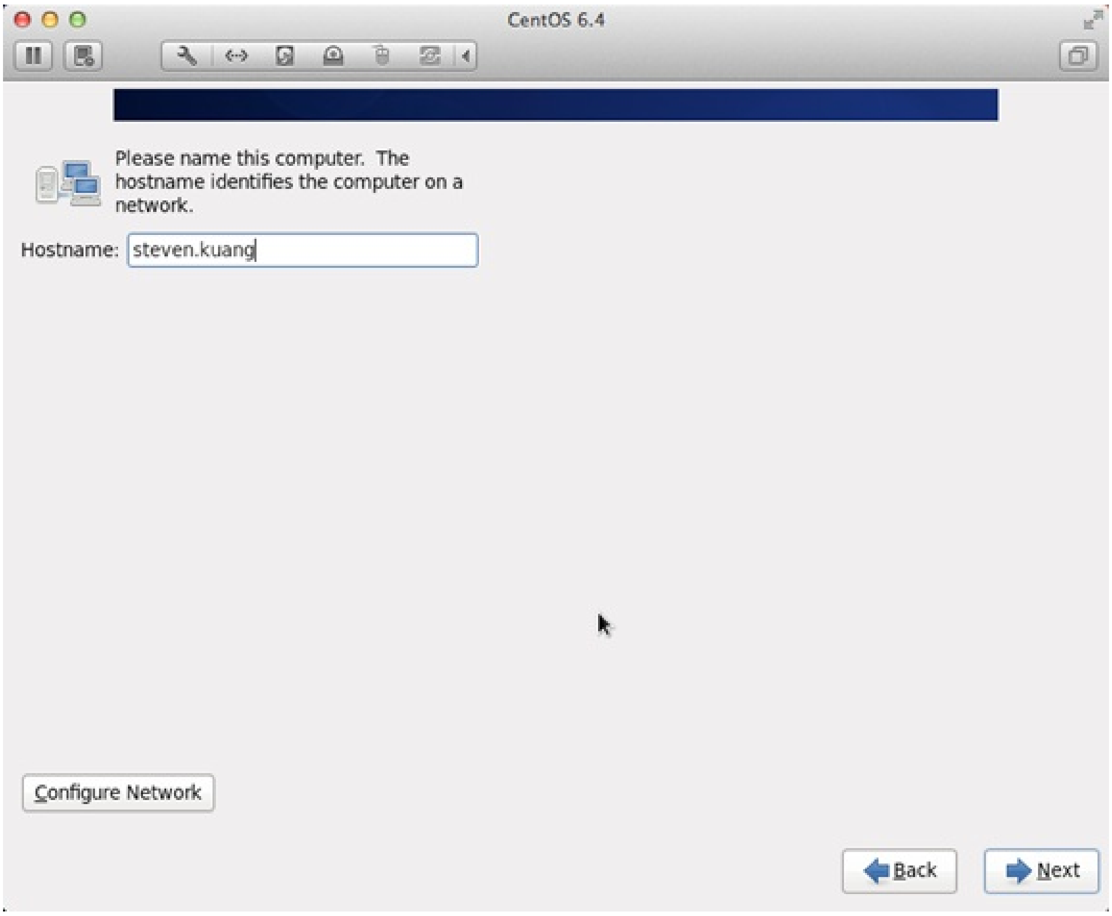
6、选择"Basic Storage Devies"点击"Next"



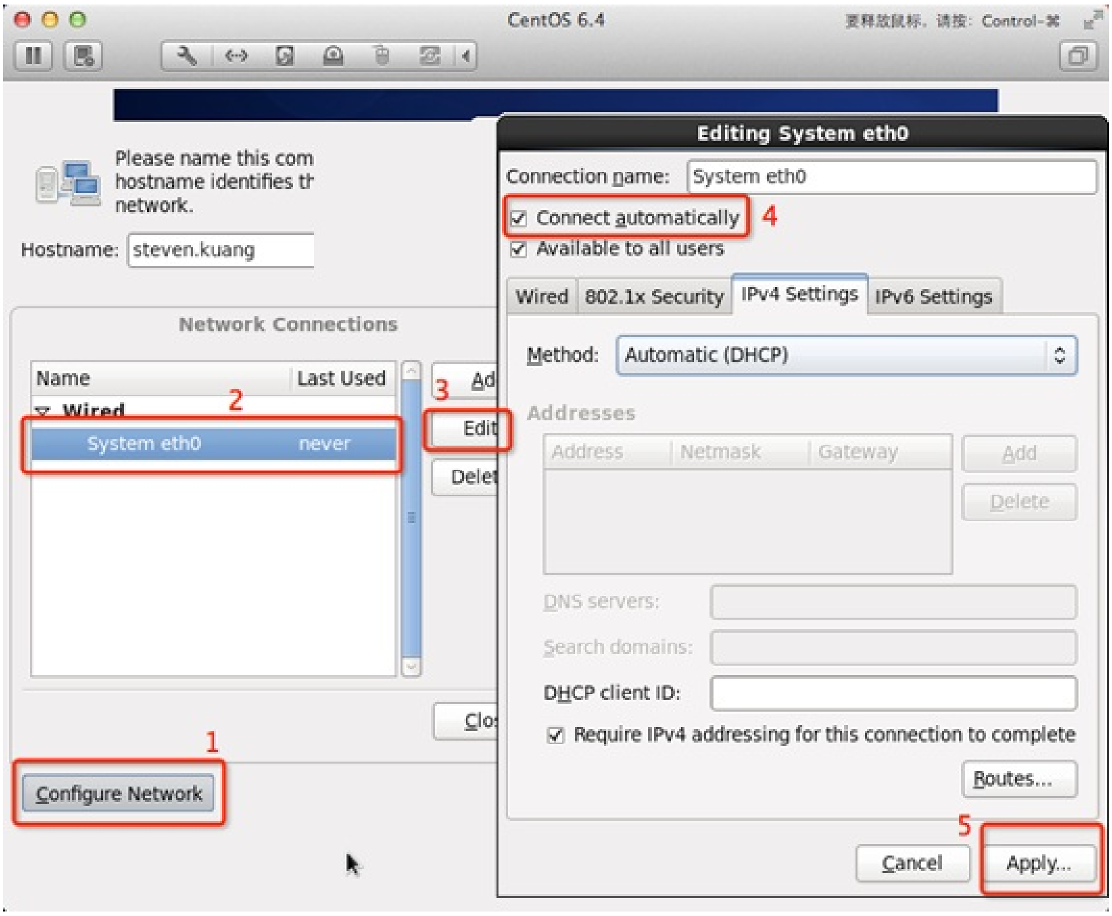
7、询问是否忽略所有数据，新电脑安装系统选择"Yes,discard any data"



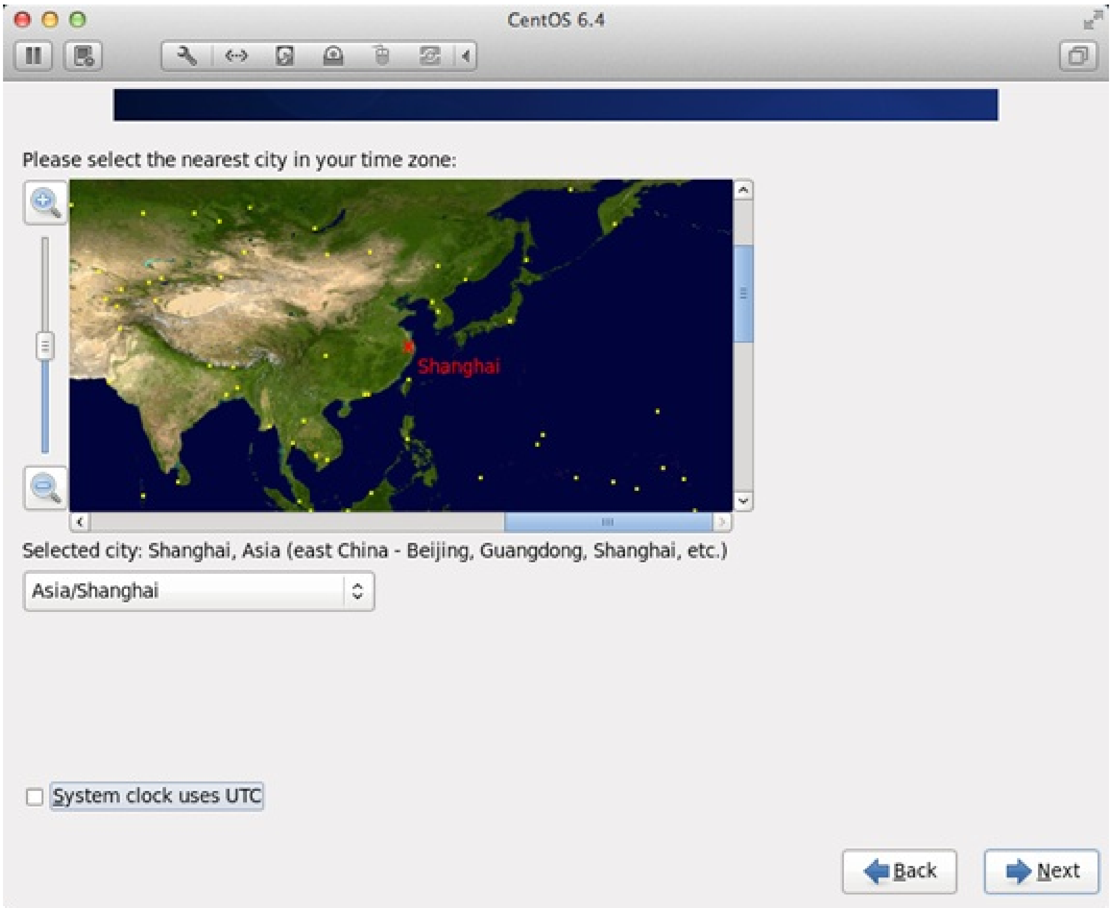
8、Hostname填写格式"英文名.姓"



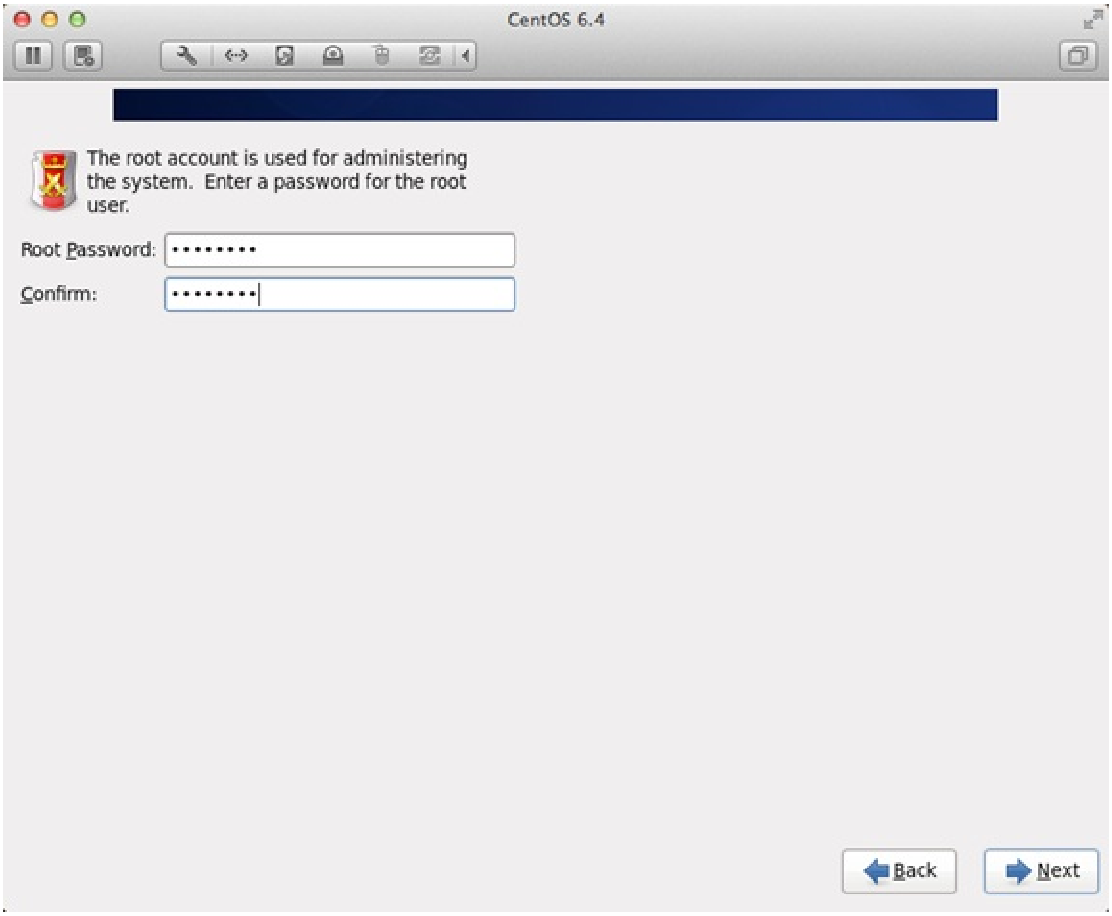
9、网络设置安装图示顺序点击就可以了



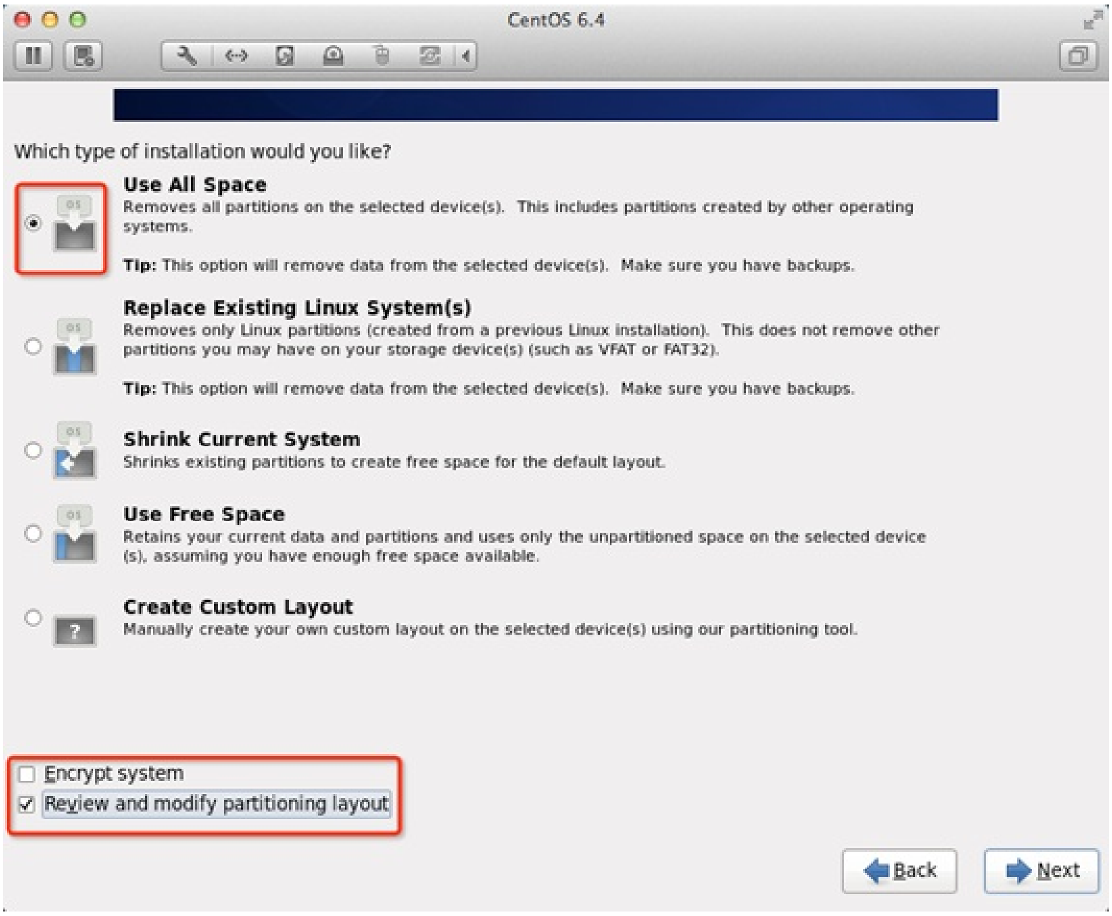
10、 时区可以在地图上点击，选择"shanghai"并取消System clock uses UTC前面的对勾



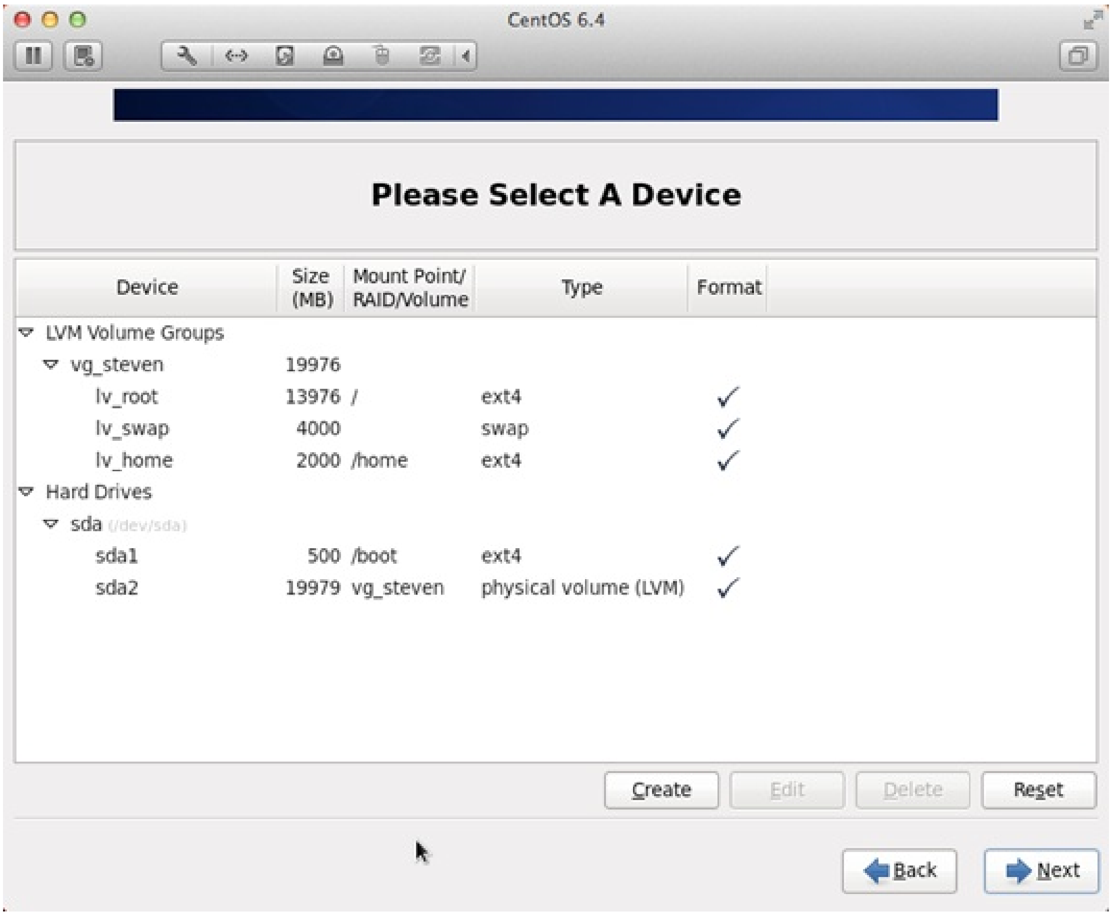
11、设置root的密码



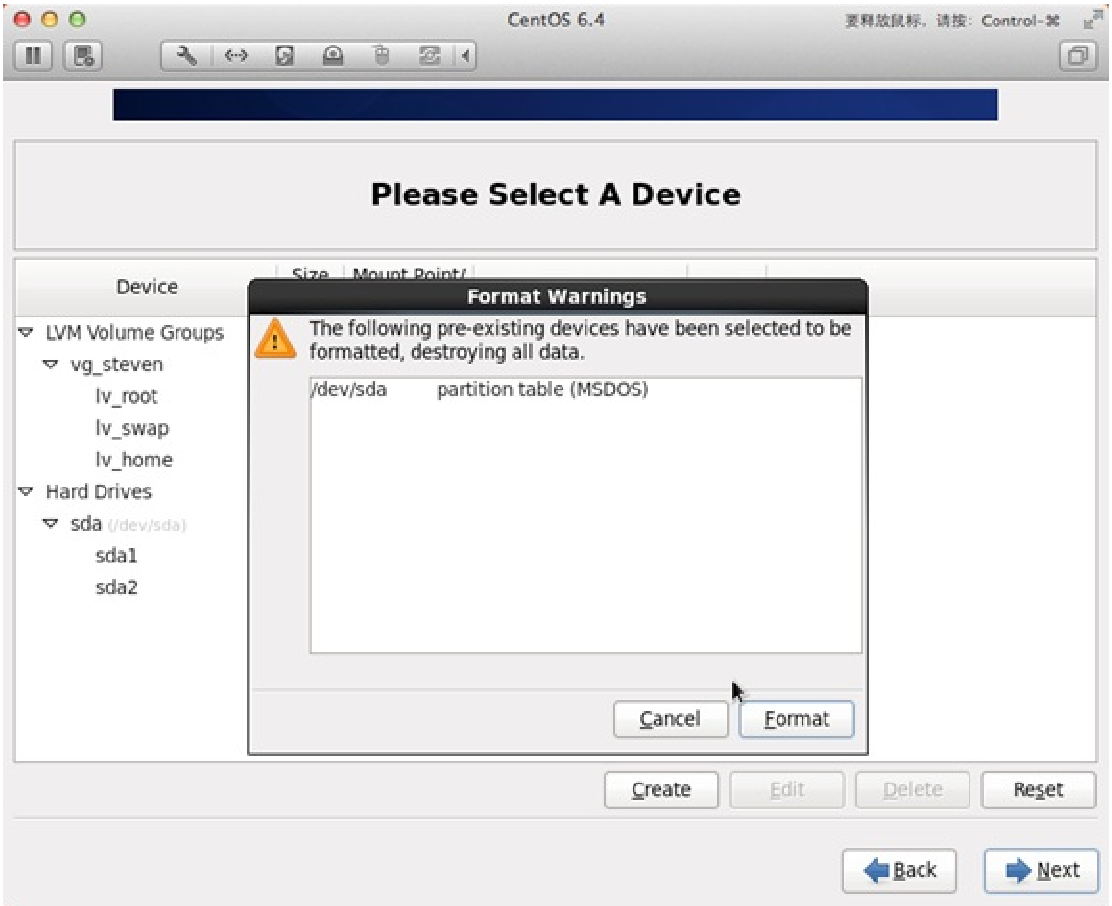
12、硬盘分区，一定要按照图示点选



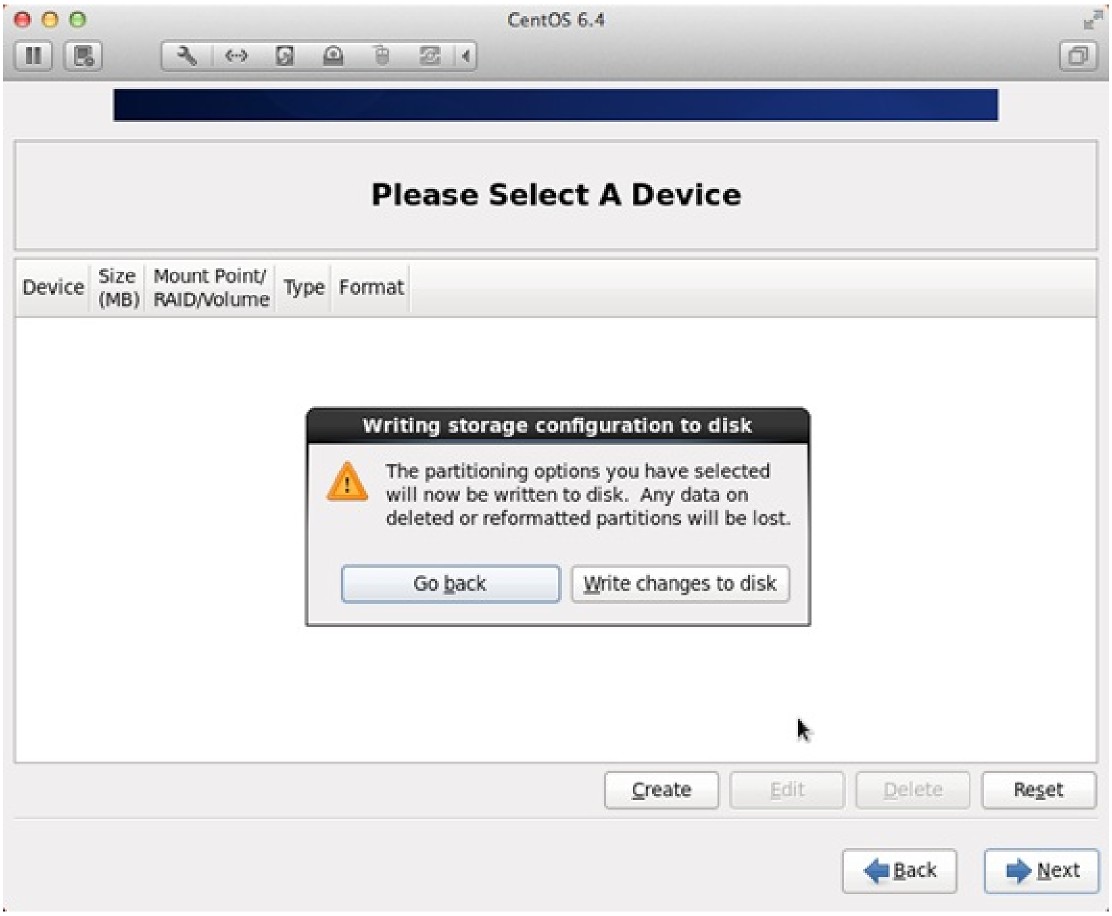
13、调整分区，必须要有/home这个分区，如果没有这个分区，安装部分软件会出现不能安装的问题



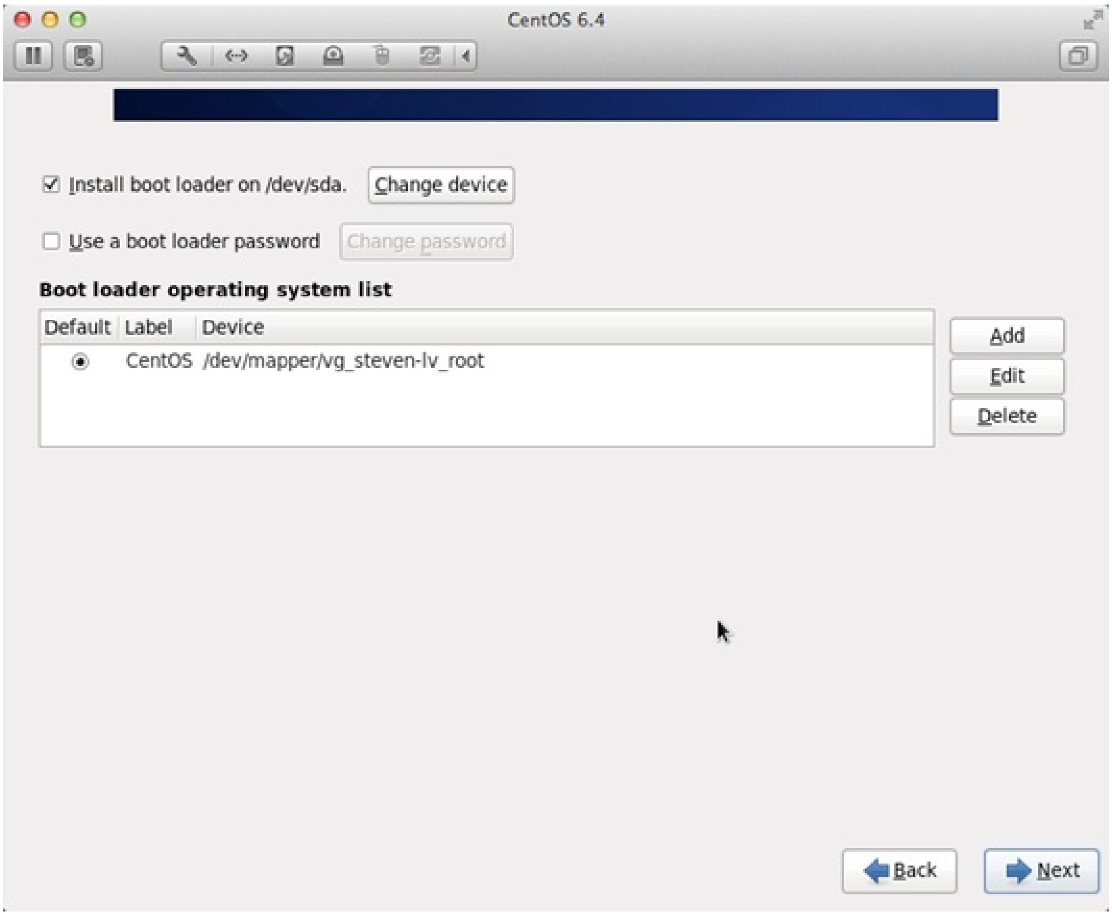
14、询问是否格式化分区



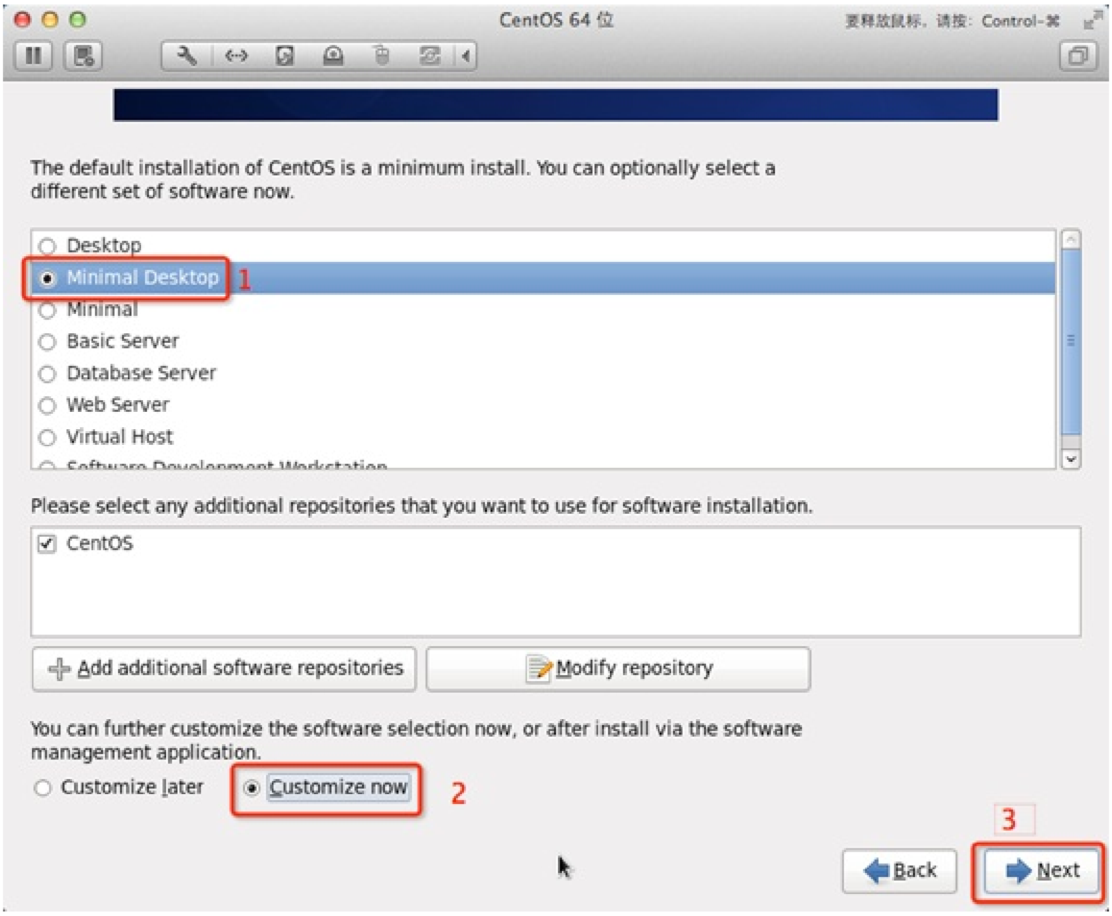
15、将更改写入到硬盘



16、引导程序安装位置



17、最重要的一步，也是本教程最关机的一步，也是其他教程没有提及的一步，按图示顺序点击



18、取消以下内容的所有选项

Applications

Base System

Servers

并对Desktops进行如下设置

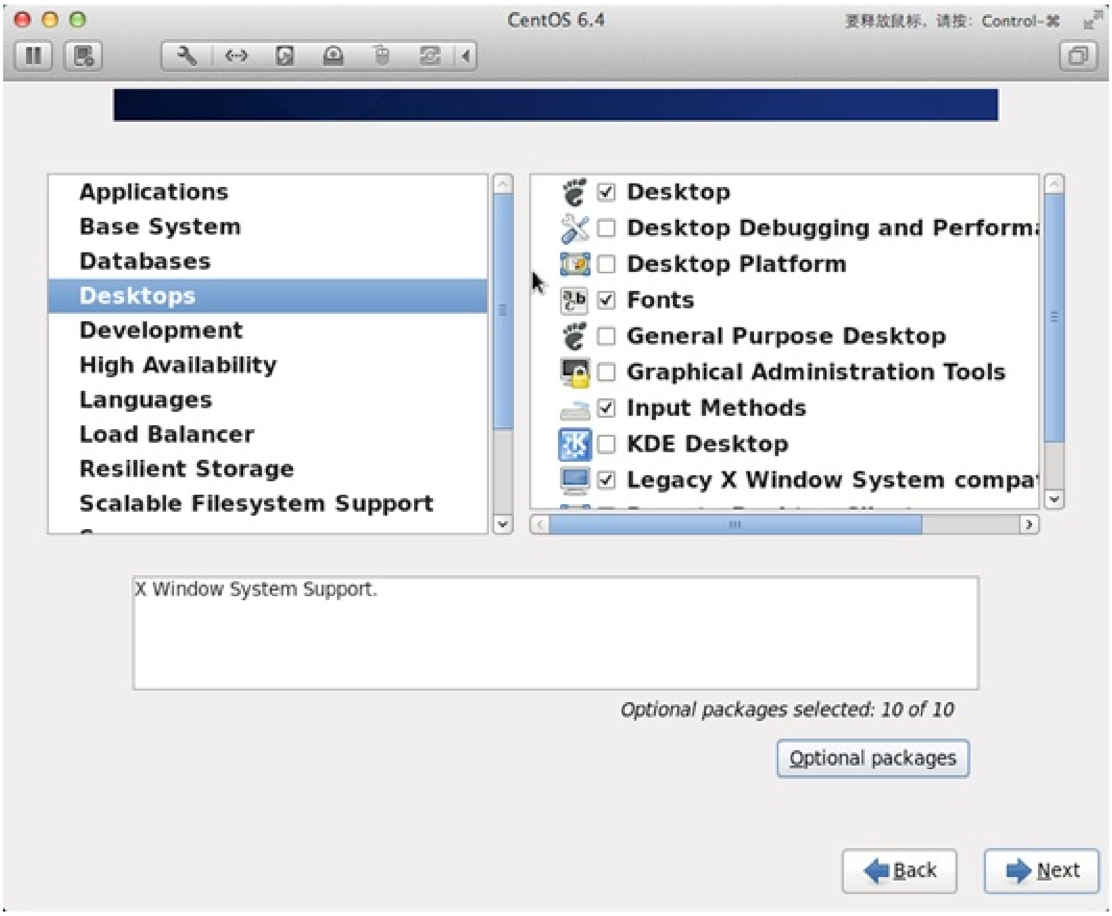
即取消如下选项：

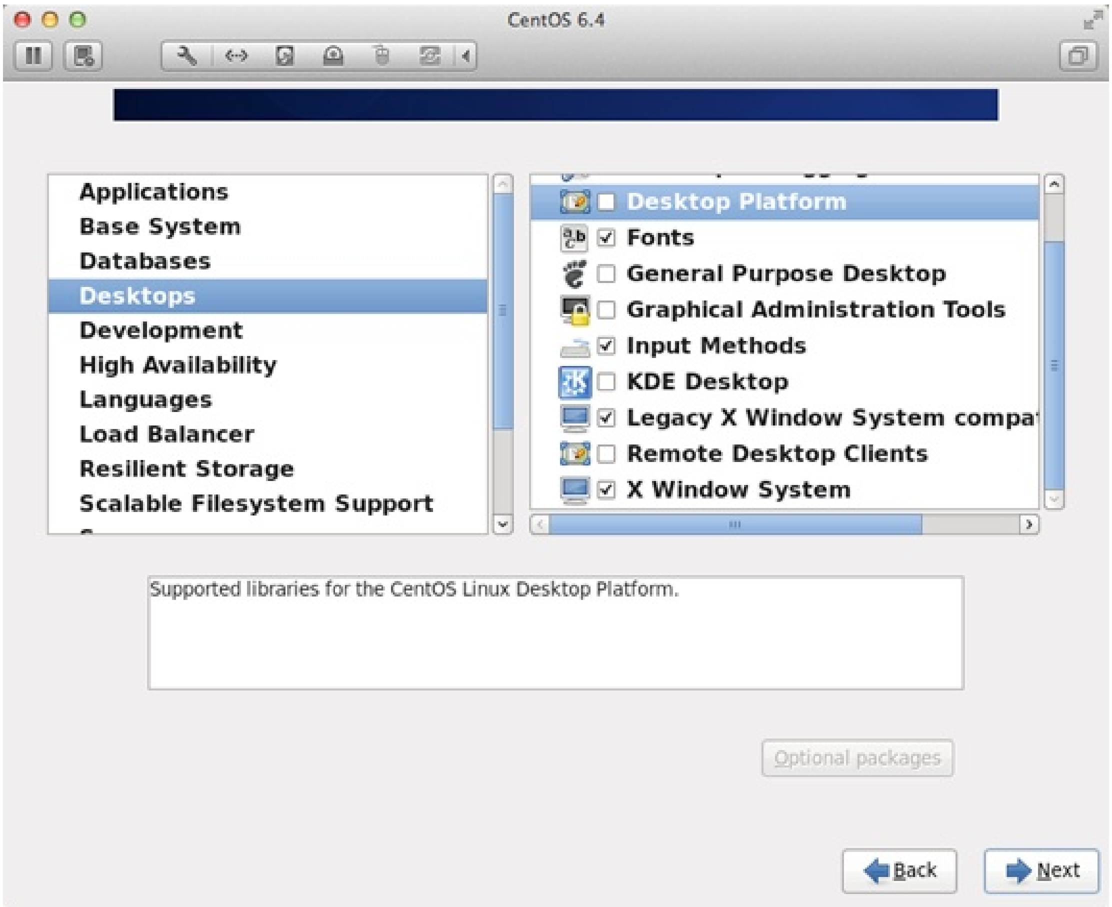
Desktop Debugging and Performance Tools

Desktop Platform

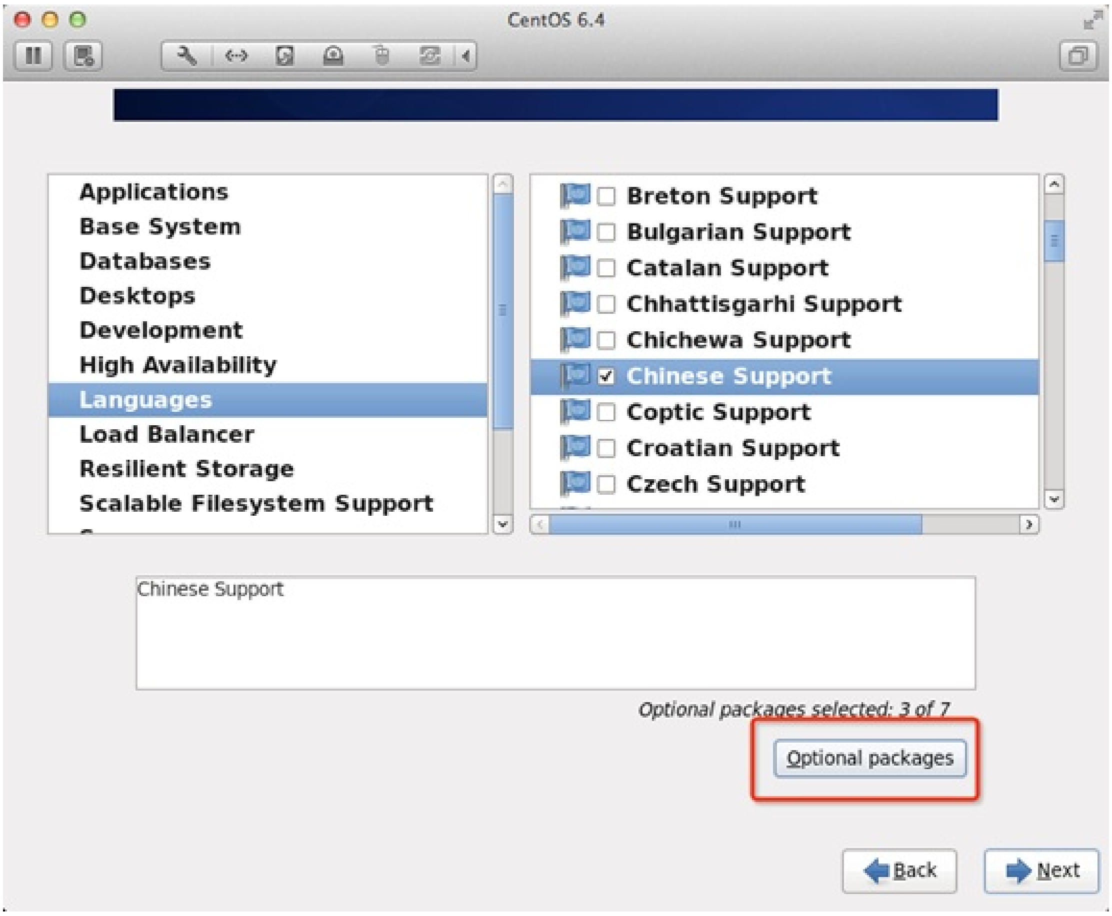
Remote Desktop Clients

Input Methods中仅保留ibus-pinyin-1.3.8-1.el6.x86_64,其他的全部取消**

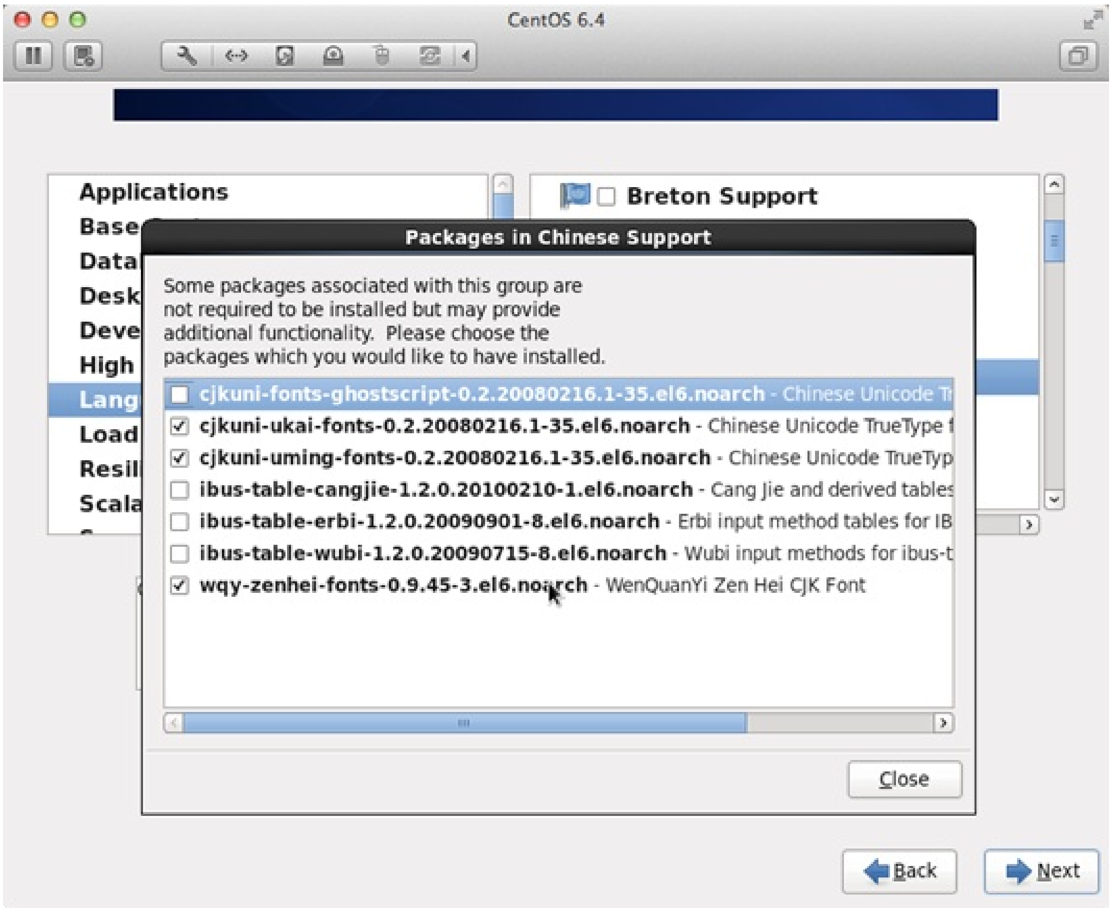




19、选中Languages，并选中右侧的Chinese Support然后点击红色区域

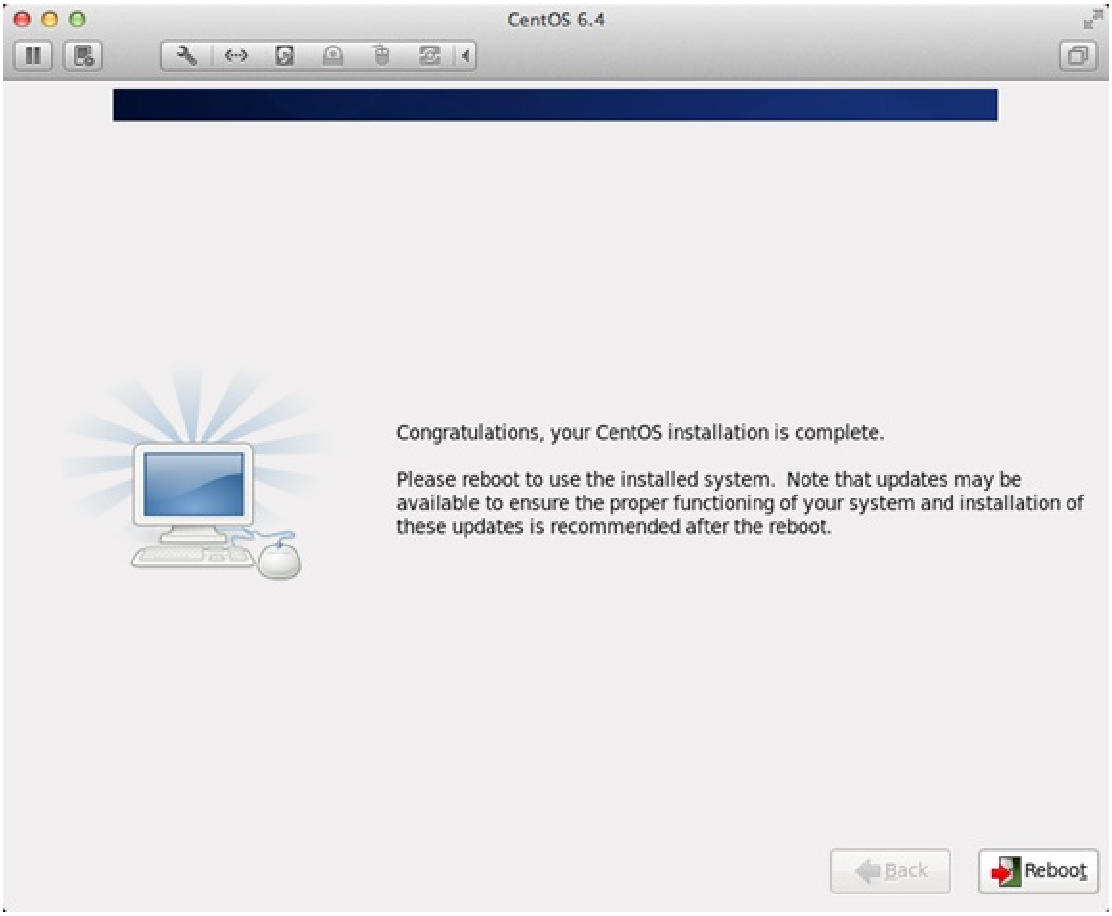


20、调整完成后如下图所示

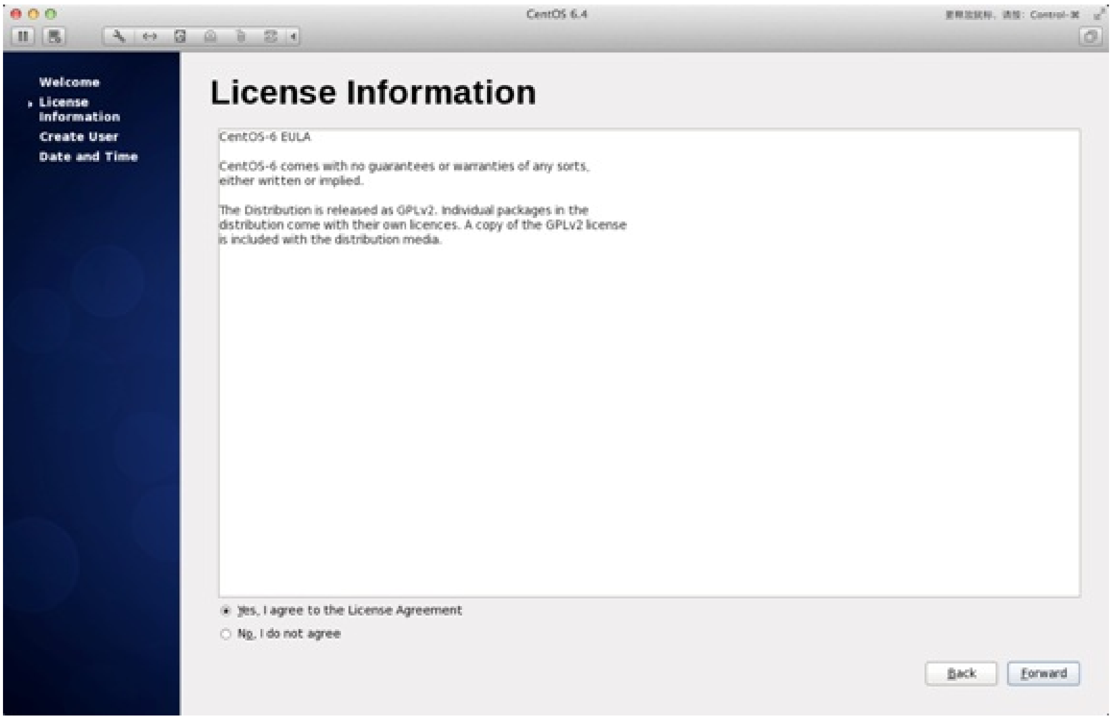


21、至此，一个最精简的桌面环境就设置完成了，

22、安装完成，重启



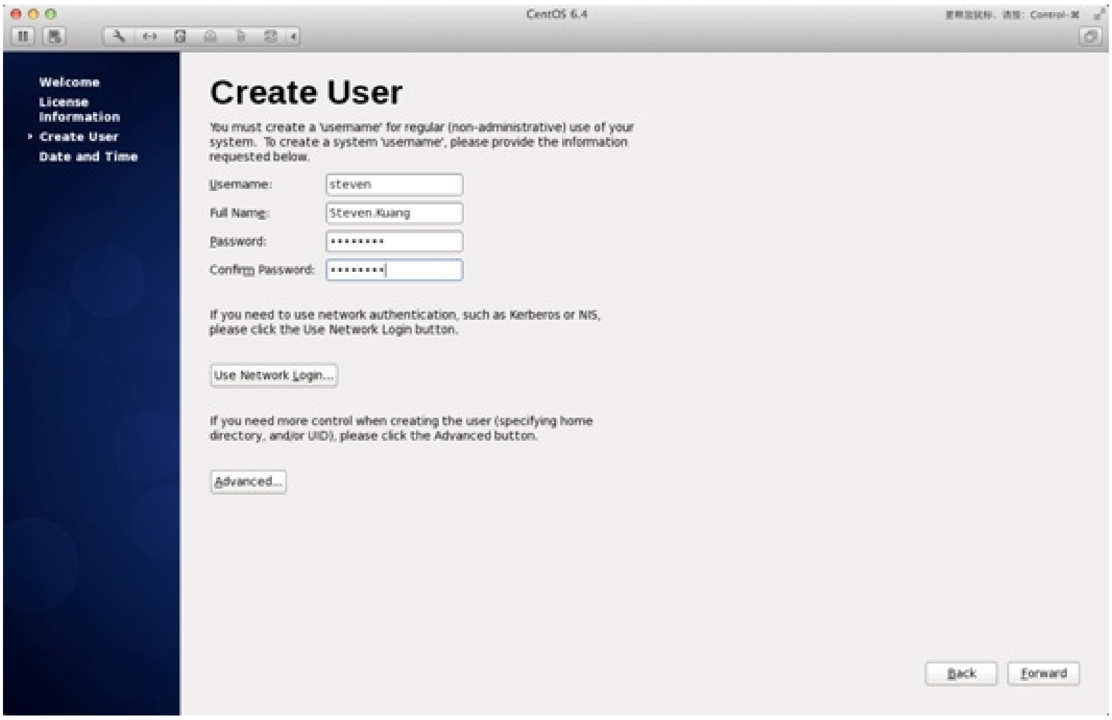
23、重启之后，的License Information



24、Create User

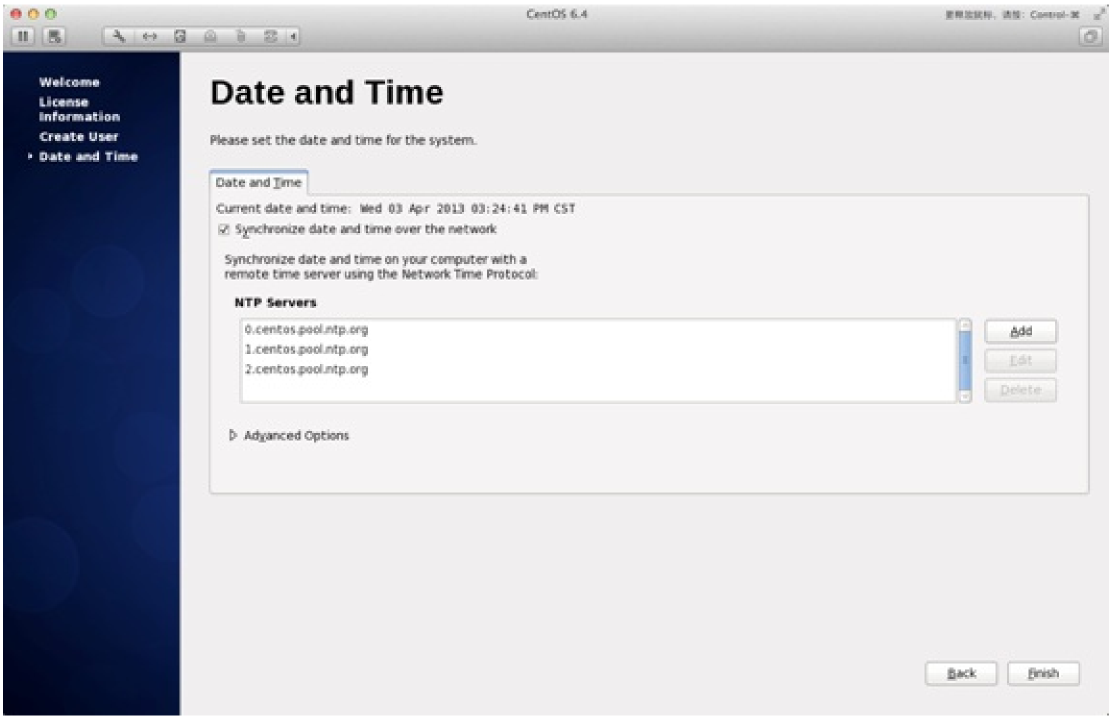
Username：填写您的英文名（不带.姓）

Full Name：填写您的英文名.姓（首字母大写）



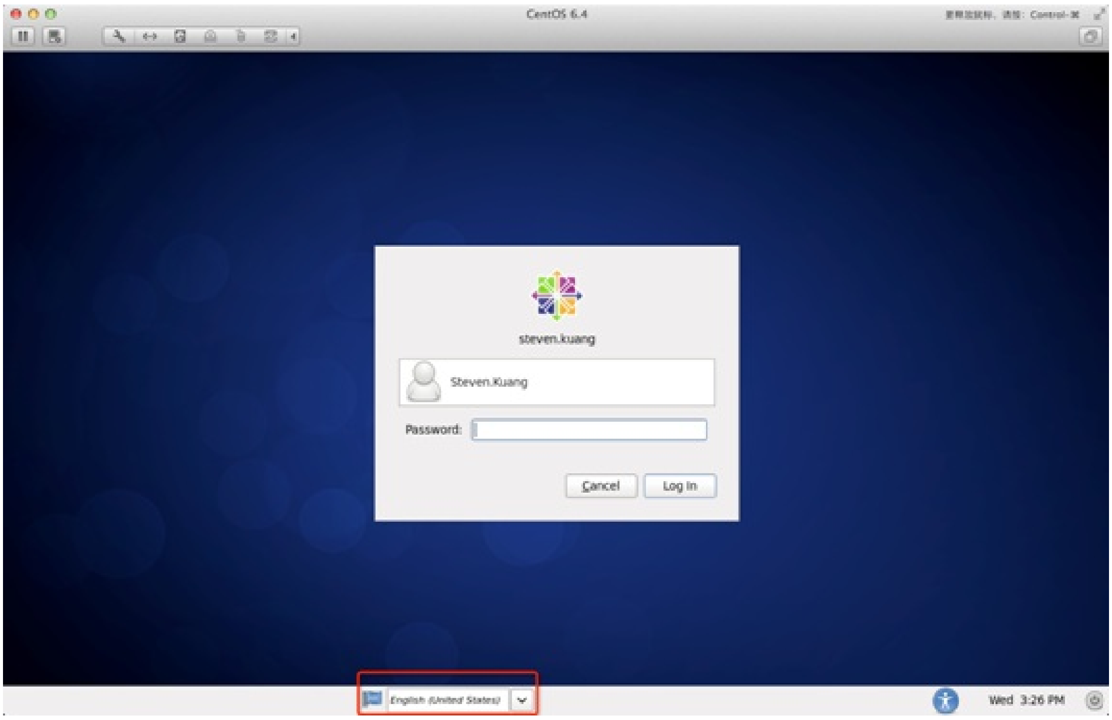
25、"Date and Time" 选中 "Synchronize data and time over the network"

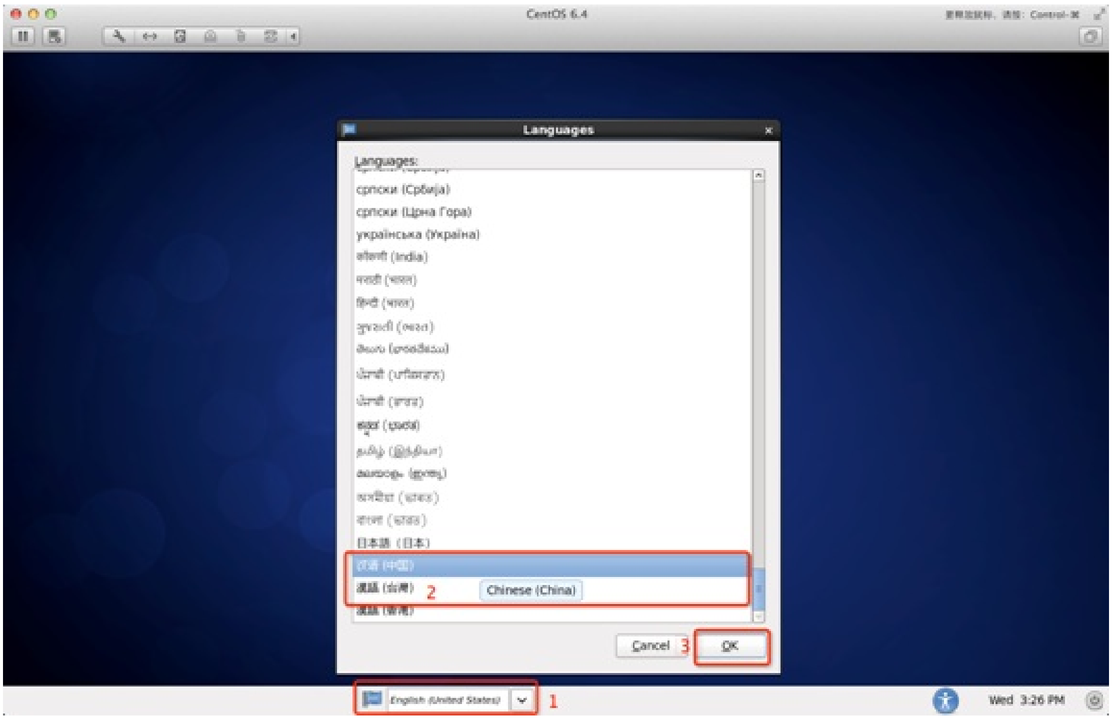
Finsh之后系统将重启



26、第一次登录，登录前不要做任何更改，这个很重要！！！登录之后紧接着退出

第二次登录，选择语言，在红色区域选择下拉小三角，选other，选中"汉语（中国）"





27、登录之后，请一定按照如下顺序点击！

至此，CentOS安装完成，如有其他问题，请随时与我联系！！

CentOS 6.4
应用程序 位置 系统
Steven.Kuang
计算机
steven 的主文件夹
回收站
将标准文件夹更新到当前语言吗
当前文件夹名称
新文件夹名称
/home/steven/Desktop
/home/steven/桌面
/home/steven/Downloads
/home/steven/下载
/home/steven/Templates
/home/steven/模板
/home/steven/Public
/home/steven/公共的
/home/steven/Documents
/home/steven/文档
/home/steven/Music
/home/steven/音乐
/home/steven/Pictures
/home/steven/图片
/home/steven/Videos
/home/steven/视频
1
不要再问我（D）
2
保留旧名称（K）
更新名称（U）

Linux 系统启动过程

linux启动时我们会看到许多启动信息。

Linux系统的启动过程并不是大家想象中的那么复杂，其过程可以分为5个阶段：

- 内核的引导。
- 运行init。
- 系统初始化。
- 建立终端 。
- 用户登录系统。

内核引导

当计算机打开电源后，首先是BIOS开机自检，按照BIOS中设置的启动设备（通常是硬盘）来启动。

操作系统接管硬件以后，首先读入 /boot 目录下的内核文件。

运行init

init 进程是系统所有进程的起点，你可以把它比拟成系统所有进程的老祖宗，没有这个进程，系统中任何进程都不会启动。

init 程序首先是需要读取配置文件 /etc/inittab。

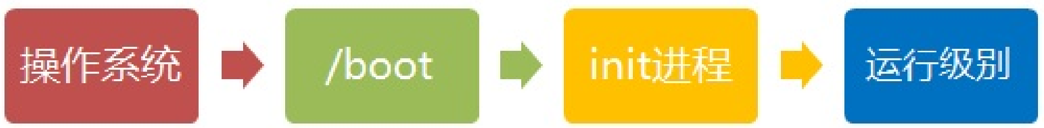
运行级别

许多程序需要开机启动。它们在Windows叫做"服务"（service），在Linux就叫做"守护进程"（daemon）。

init进程的一大任务，就是去运行这些开机启动的程序。

但是，不同的场合需要启动不同的程序，比如用作服务器时，需要启动Apache，用作桌面就不需要。

Linux允许为不同的场合，分配不同的开机启动程序，这就叫做"运行级别"（runlevel）。也就是说，启动时根据"运行级别"，确定要运行哪些程序。



Linux系统有7个运行级别(runlevel)：

- 运行级别0：系统停机状态，系统默认运行级别不能设为0，否则不能正常启动
- 运行级别1：单用户工作状态，root权限，用于系统维护，禁止远程登陆
- 运行级别2：多用户状态(没有NFS)
- 运行级别3：完全的多用户状态(有NFS)，登陆后进入控制台命令行模式
- 运行级别4：系统未使用，保留
- 运行级别5：X11控制台，登陆后进入图形GUI模式
- 运行级别6：系统正常关闭并重启，默认运行级别不能设为6，否则不能正常启动

系统初始化

在init的配置文件中有这么一行： si::sysinit:/etc/rc.d/rc.sysinit　它调用执行了/etc/rc.d/rc.sysinit，而rc.sysinit是一个bash shell的脚本，它主要是完成一些系统初始化的工作，rc.sysinit是每一个运行级别都要首先运行的重要脚本。

它主要完成的工作有：激活交换分区，检查磁盘，加载硬件模块以及其它一些需要优先执行任务。

```
l5:5:wait:/etc/rc.d/rc 5
```

这一行表示以5为参数运行/etc/rc.d/rc，/etc/rc.d/rc是一个Shell脚本，它接受5作为参数，去执行/etc/rc.d/rc5.d/目录下的所有的rc启动脚本，/etc/rc.d/rc5.d/目录中的这些启动脚本实际上都是一些连接文件，而不是真正的rc启动脚本，真正的rc启动脚本实际上都是放在/etc/rc.d/init.d/目录下。

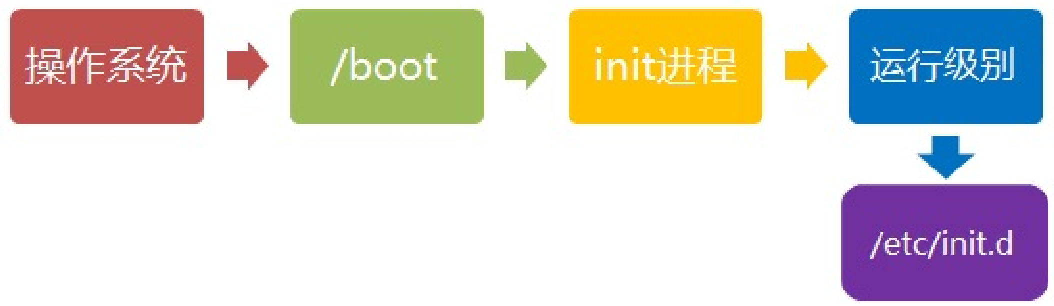
而这些rc启动脚本有着类似的用法，它们一般能接受start、stop、restart、status等参数。

/etc/rc.d/rc5.d/中的rc启动脚本通常是K或S开头的连接文件，对于以以S开头的启动脚本，将以start参数来运行。

而如果发现存在相应的脚本也存在K打头的连接，而且已经处于运行态了(以/var/lock/subsys/下的文件作为标志)，则将首先以stop为参数停止这些已经启动了的守护进程，然后再重新运行。

这样做是为了保证是当init改变运行级别时，所有相关的守护进程都将重启。

至于在每个运行级中将运行哪些守护进程，用户可以通过chkconfig或setup中的"System Services"来自行设定。



建立终端

rc执行完毕后，返回init。这时基本系统环境已经设置好了，各种守护进程也已经启动了。

init接下来会打开6个终端，以便用户登录系统。在inittab中的以下6行就是定义了6个终端：

```
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

从上面可以看出在2、3、4、5的运行级别中都将以respawn方式运行mingetty程序，mingetty程序能打开终端、设置模式。

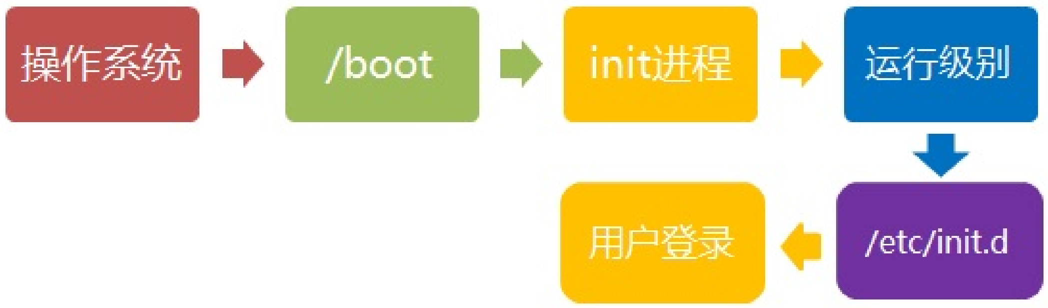
同时它会显示一个文本登录界面，这个界面就是我们经常看到的登录界面，在这个登录界面中会提示用户输入用户名，而用户输入的用户将作为参数传给login程序来验证用户的身份。

用户登录系统

一般来说，用户的登录方式有三种：

- （1）命令行登录

- （2）ssh登录
- （3）图形界面登录



对于运行级别为5的图形方式用户来说，他们的登录是通过一个图形化的登录界面。登录成功后可以直接进入KDE、Gnome等窗口管理器。

而本文主要讲的还是文本方式登录的情况：当我们看到mingetty的登录界面时，我们就可以输入用户名和密码来登录系统了。

Linux的账号验证程序是login，login会接收mingetty传来的用户名作为用户名参数。

然后login会对用户名进行分析：如果用户名不是root，且存在/etc/nologin文件，login将输出nologin文件的内容，然后退出。

这通常用来系统维护时防止非root用户登录。只有/etc/securetty中登记了的终端才允许root用户登录，如果不存在这个文件，则root可以在任何终端上登录。

/etc/usertty文件用于对用户作出附加访问限制，如果不存在这个文件，则没有其他限制。

图形模式与文字模式的切换方式

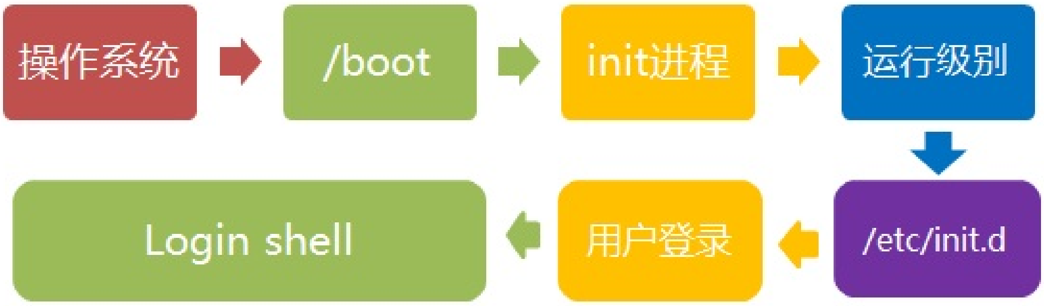
Linux预设提供了六个命令窗口终端机让我们来登录。

默认我们登录的就是第一个窗口，也就是tty1，这个六个窗口分别为tty1,tty2 … tty6，你可以按下Ctrl + Alt + F1 ~ F6 来切换它们。

如果你安装了图形界面，默认情况下是进入图形界面的，此时你就可以按Ctrl + Alt + F1 ~ F6来进入其中一个命令窗口界面。

当你进入命令窗口界面后再返回图形界面只要按下Ctrl + Alt + F7 就回来了。

如果你用的vmware 虚拟机，命令窗口切换的快捷键为 Alt + Space + F1~F6. 如果你在图形界面下请按Alt + Shift + Ctrl + F1~F6 切换至命令窗口。



Linux 关机

在linux领域内大多用在服务器上，很少遇到关机的操作。毕竟服务器上跑一个服务是永无止境的，除非特殊情况下，不得已才会关机。

正确的关机流程为：sysnc > shutdown > reboot > halt

关机指令为：shutdown，你可以man shutdown 来看一下帮助文档。

例如你可以运行如下命令关机：

```
sync 将数据由内存同步到硬盘中。
shutdown 关机指令，你可以man shutdown 来看一下帮助文档。例如你可以运行如下命令关机：
shutdown –h 10 ‘This server will shutdown after 10 mins’ 这个命令告诉大家，计算机将在10分钟后关
Shutdown –h now 立马关机
Shutdown –h 20:25 系统会在今天20:25关机
Shutdown –h +10 十分钟后关机
Shutdown –r now 系统立马重启
Shutdown –r +10 系统十分钟后重启
reboot 就是重启，等同于 shutdown –r now
halt 关闭系统，等同于shutdown –h now 和 poweroff
```

最后总结一下，不管是重启系统还是关闭系统，首先要运行sync命令，把内存中的数据写到磁盘中。

关机的命令有 shutdown –h now halt poweroff 和 init 0，重启系统的命令有 shutdown –r now reboot init 6.

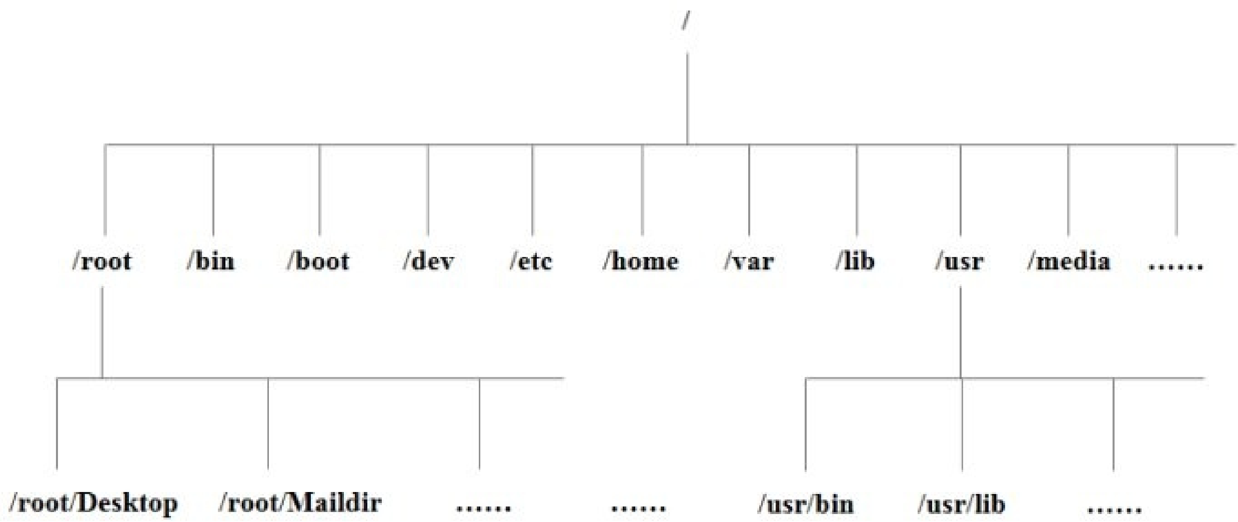
Linux 系统目录结构

登录系统后，在当前命令窗口下输入命令：

```
ls /
```

你会看到如下图所示:

树状目录结构：



以下是对这些目录的解释：

- **/bin**：
  bin是Binary的缩写, 这个目录存放着最经常使用的命令。
- **/boot**：
  这里存放的是启动Linux时使用的一些核心文件，包括一些连接文件以及镜像文件。
- **/dev** ：
  dev是Device(设备)的缩写, 该目录下存放的是Linux的外部设备，在Linux中访问设备的方式和访问文件的方式是相同的。
- **/etc**：
  这个目录用来存放所有的系统管理所需要的配置文件和子目录。
- **/home**：
  用户的主目录，在Linux中，每个用户都有一个自己的目录，一般该目录名是以用户的账号命名的。

- **/lib**：
  这个目录里存放着系统最基本的动态连接共享库，其作用类似于Windows里的DLL文件。几乎所有的应用程序都需要用到这些共享库。

- **/lost+found**：
  这个目录一般情况下是空的，当系统非法关机后，这里就存放了一些文件。

- /media linux系统会自动识别一些设备，例如U盘、光驱等等，当识别后，linux会把识别的设备挂载到这个目录下。

- **/mnt**：
  系统提供该目录是为了让用户临时挂载别的文件系统的，我们可以将光驱挂载在/mnt/上，然后进入该目录就可以查看光驱里的内容了。

- **/opt**：
  这是给主机额外安装软件所摆放的目录。比如你安装一个ORACLE数据库则就可以放到这个目录下。默认是空的。

- **/proc**：
  这个目录是一个虚拟的目录，它是系统内存的映射，我们可以通过直接访问这个目录来获取系统信息。
  这个目录的内容不在硬盘上而是在内存里，我们也可以直接修改里面的某些文件，比如可以通过下面的命令来屏蔽主机的ping命令，使别人无法ping你的机器：

  ```
  echo 1 &gt; /proc/sys/net/ipv4/icmp_echo_ignore_all
  ```

- **/root**：
  该目录为系统管理员，也称作超级权限者的用户主目录。

- **/sbin**：
  s就是Super User的意思，这里存放的是系统管理员使用的系统管理程序。

- **/selinux**：
  这个目录是Redhat/CentOS所特有的目录，Selinux是一个安全机制，类似于windows的防火墙，但是这套机制比较复杂，这个目录就是存放selinux相关的文件的。

- **/srv**：
  该目录存放一些服务启动之后需要提取的数据。

- **/sys**：
  这是linux2.6内核的一个很大的变化。该目录下安装了2.6内核中新出现的一个文件系统 sysfs 。

  sysfs文件系统集成了下面3种文件系统的信息：针对进程信息的proc文件系统、针对设备的devfs文件系统以及针对伪终端的devpts文件系统。

该文件系统是内核设备树的一个直观反映。

当一个内核对象被创建的时候，对应的文件和目录也在内核对象子系统种被创建。

- **/tmp**：
这个目录是用来存放一些临时文件的。

- **/usr**：
这是一个非常重要的目录，用户的很多应用程序和文件都放在这个目录下，类似与windows下的program files目录。

- **/usr/bin**：
系统用户使用的应用程序。

- **/usr/sbin**：
超级用户使用的比较高级的管理程序和系统守护程序。

- **/usr/src**：内核源代码默认的放置目录。

- **/var**：
这个目录中存放着在不断扩充着的东西，我们习惯将那些经常被修改的目录放在这个目录下。包括各种日志文件。

在linux系统中，有几个目录是比较重要的，平时需要注意不要误删除或者随意更改内部文件。

/etc： 上边也提到了，这个是系统中的配置文件，如果你更改了该目录下的某个文件可能会导致系统不能启动。

/bin, /sbin, /usr/bin, /usr/sbin: 这是系统预设的执行文件的放置目录，比如 ls 就是在/bin/ls 目录下的。

值得提出的是，/bin, /usr/bin 是给系统用户使用的指令（除root外的通用户），而/sbin, /usr/sbin 则是给root使用的指令。

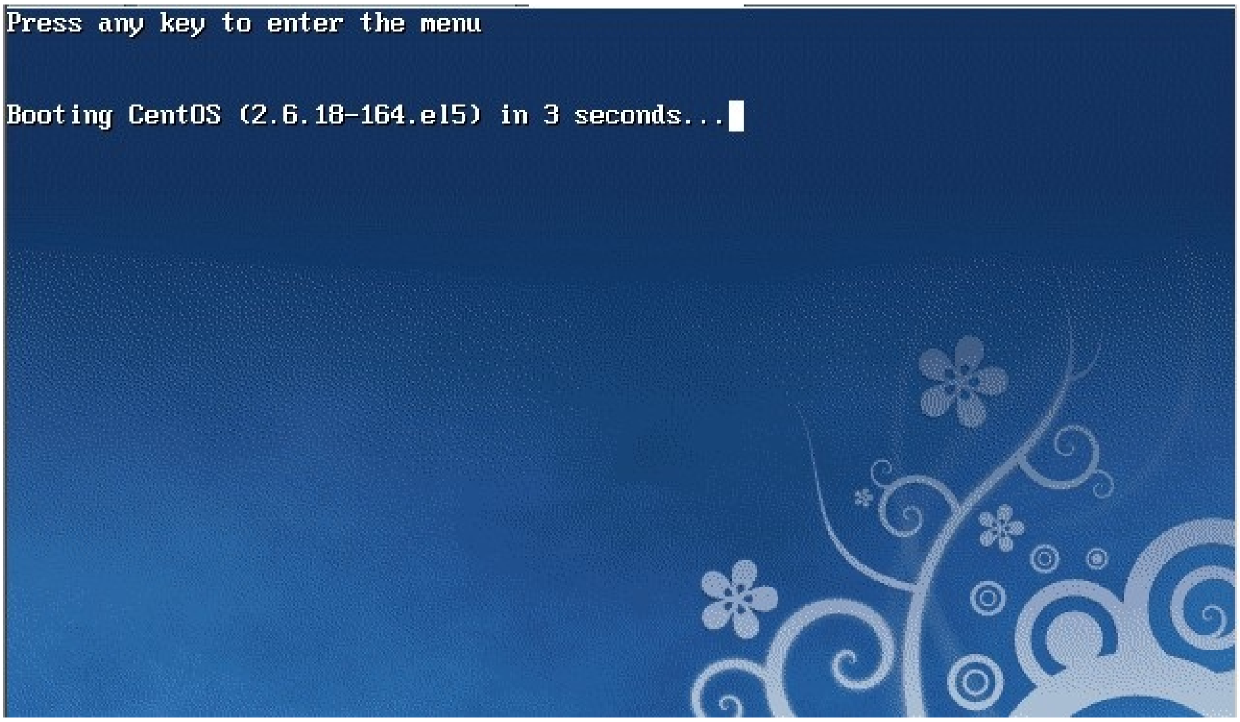
/var： 这是一个非常重要的目录，系统上跑了很多程序，那么每个程序都会有相应的日志产生，而这些日志就被记录到这个目录下，具体在/var/log 目录下，另外mail的预设放置也是在这里。

Linux 忘记密码解决方法

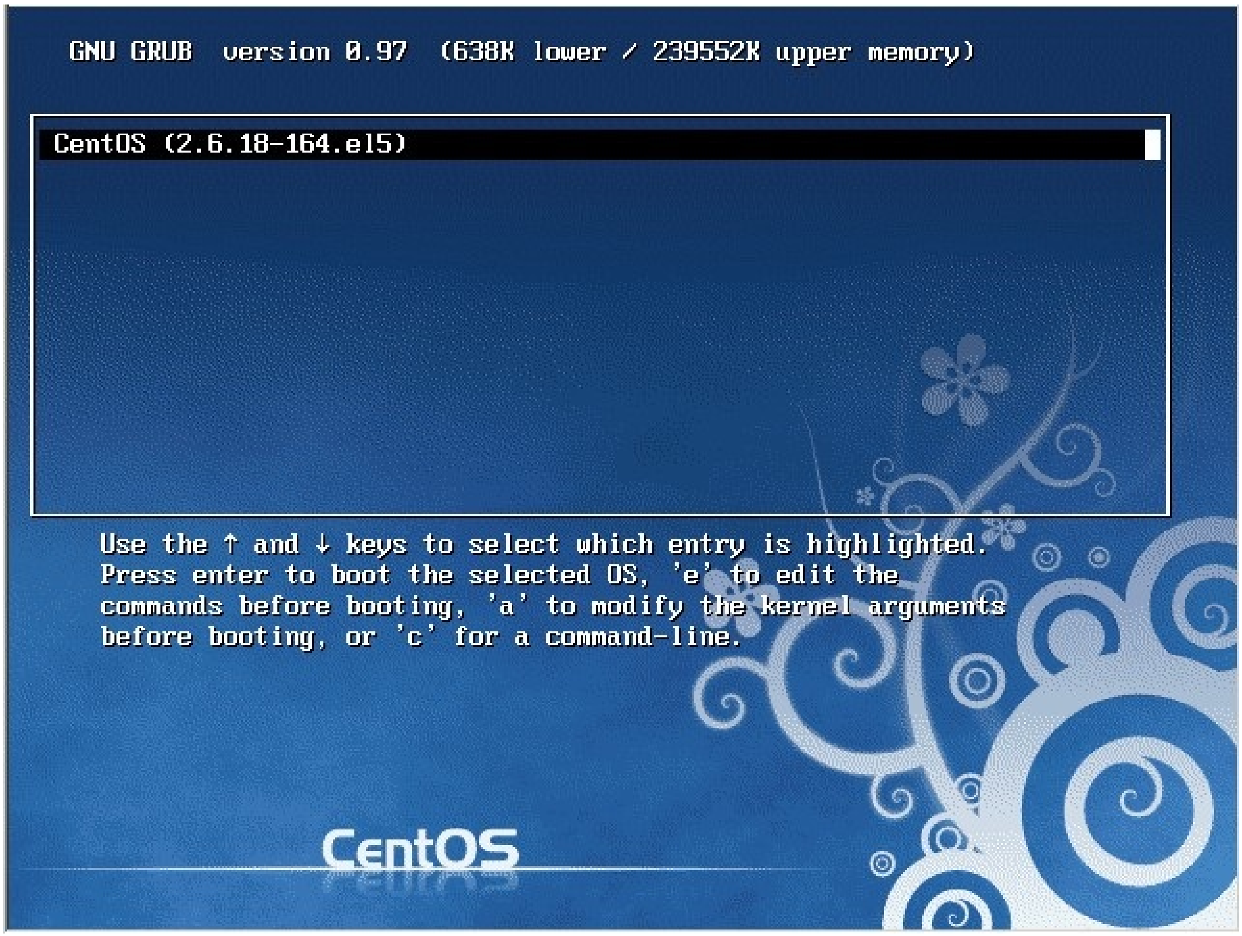
很多朋友经常会忘记Linux系统的root密码，linux系统忘记root密码的情况该怎么办呢？重新安装系统吗？当然不用！进入单用户模式更改一下root密码即可。

步骤如下：

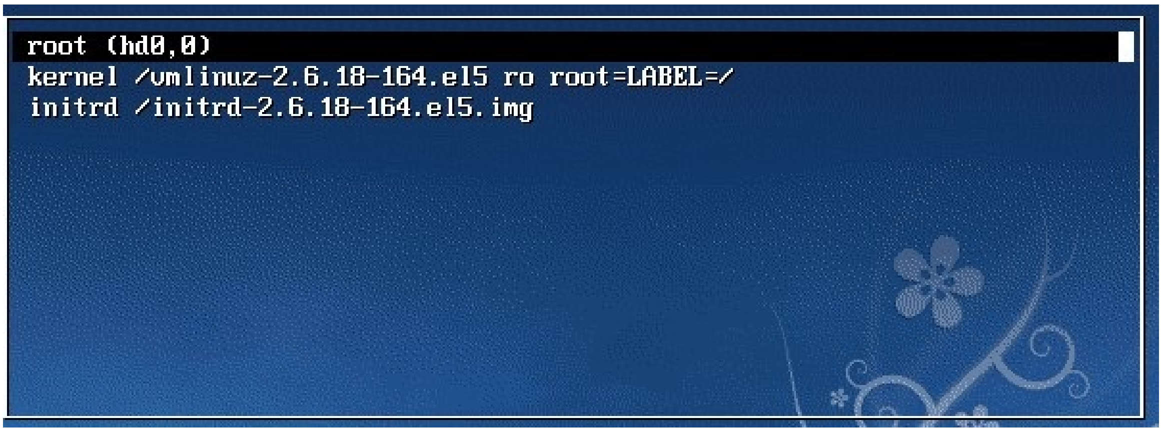
重启linux系统



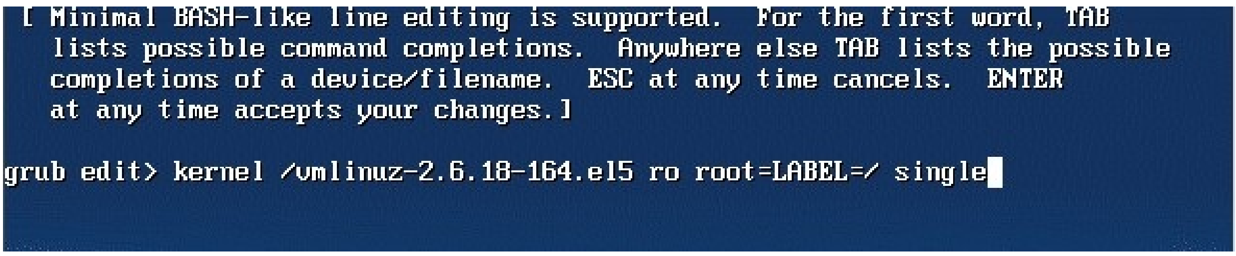
3 秒之内要按一下回车，出现如下界面



然后输入e



在 第二行最后边输入 single，有一个空格。具体方法为按向下尖头移动到第二行，按"e"进入编辑模式



在后边加上single 回车

```
root (hd0,0)
kernel /vmlinuz-2.6.18-164.el5 ro root=LABEL=/ single
initrd /initrd-2.6.18-164.el5.img
```

最后按"b"启动，启动后就进入了单用户模式了

```
no fstab.sys, mounting internal defaults
Switching to new root and running init.
unmounting old /dev
unmounting old /proc
unmounting old /sys
type=1404 audit(1303914022.636:2): enforcing=1 old_enforcing=0 auid=4294967295 s
es=4294967295
type=1403 audit(1303914023.222:3): policy loaded auid=4294967295 ses=4294967295
INIT: version 2.86 booting
                Welcome to  CentOS release 5.4 (Final)
                Press 'I' to enter interactive startup.
Setting clock  (utc): Wed Apr 27 22:20:50 CST 2011         [  OK  ]
Starting udev:                                             [  OK  ]
Loading default keymap (us):                               [  OK  ]
Setting hostname localhost.localdomain:                    [  OK  ]
No devices found
Setting up Logical Volume Management:                      [  OK  ]
Checking filesystems
/: clean, 118508/1969568 files, 713597/1967962 blocks
/boot: clean, 35/26104 files, 14714/104388 blocks
                                                           [  OK  ]
Remounting root filesystem in read-write mode:             [  OK  ]
Mounting local filesystems:                                [  OK  ]
Enabling /etc/fstab swaps:                                 [  OK  ]
sh-3.2# _
```

此时已经进入到单用户模式了，你可以更改root密码了。更密码的命令为 passwd

```
sh-3.2# passwd
Changing password for user root.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

【使用系统安装光盘的救援模式】

救援模式即rescue，这个模式主要是应用于，系统无法进入的情况。如，grub损坏或者某一个配置文件修改出错。如何使用rescue模式呢？

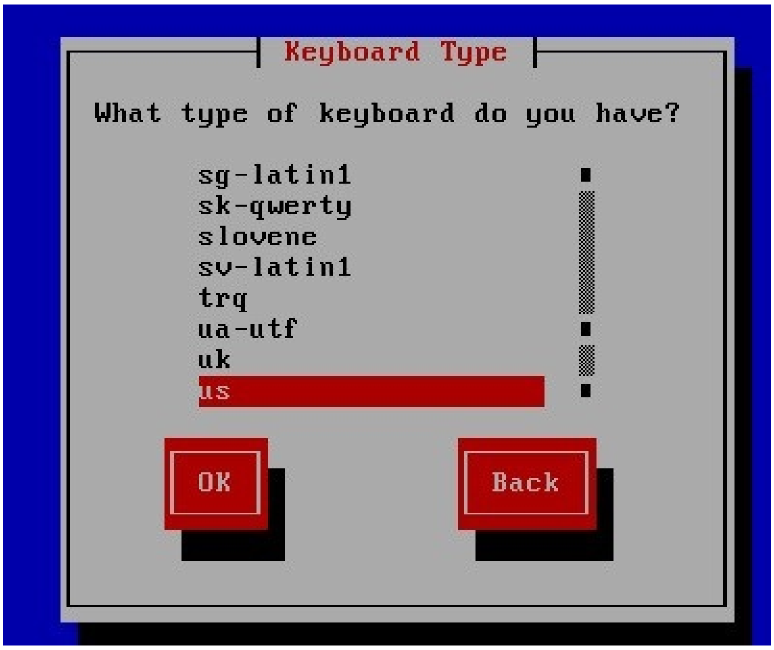
光盘启动，按F5 进入rescue模式

```
 -  To install or upgrade in graphical mode, press the <ENTER> key.

 -  To install or upgrade in text mode, type: linux text <ENTER>.

 -  Use the function keys listed below for more information.

[F1-Main] [F2-Options] [F3-General] [F4-Kernel] [F5-Rescue]
boot: _
```

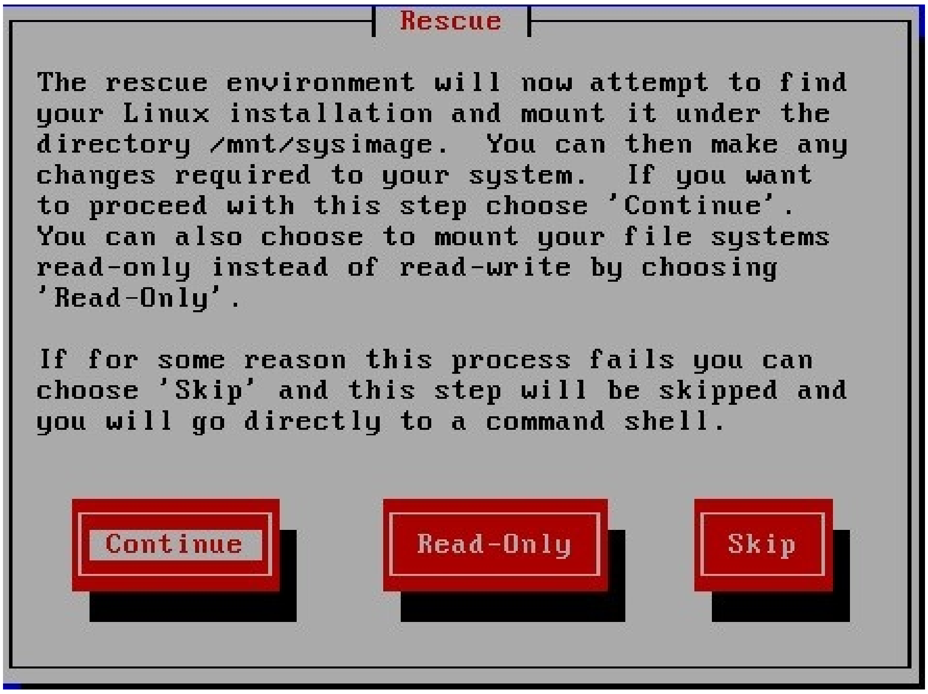
输入linux rescue 回车

选择语言，笔者建议你选择英语

选择us 键盘



这里问你是否启动网络，有时候可能会联网调试。我们选no



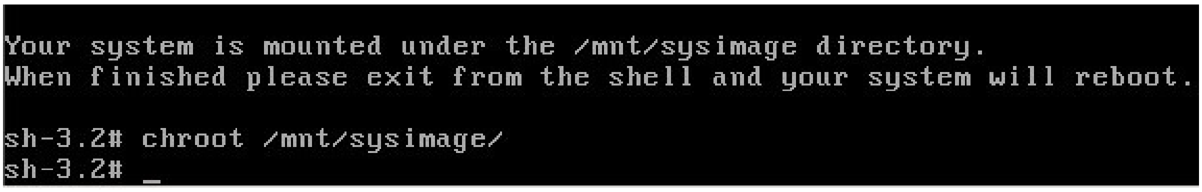
这里告诉我们，接下来会把系统挂载在/mnt/sysimage 中。

其中有三个选项:

- Continue 就是挂载后继续下一步。
- Read-Only 挂载成只读，这样更安全，有时文件系统损坏时，只读模式会防止文件系统近一步损坏。
- Skip就是不挂载，进入一个命令窗口模式。

这里我们选择Continue。

至此，系统已经挂载到了/mnt/sysimage中。接下来回车，输入chroot /mnt/sysimage 进入管理员环境。



提示： 其实也可以到rescue模式下更改root的密码的。这个rescue模式和windows PE系统很相近。

当运行了chroot /mnt/sysimage/ 后，再ls 看到目录结构和原来系统中的目录结构是一样的。

没错！现在的环境和原来系统的环境是一模一样的。你可以输入exit 或者按Ctrl + D退出这个环境。然后你再ls 看一下

这个目录其实就是rescue模式下的目录结构，而我们的系统文件全部在 /mnt/sysimage目录下。

Linux 远程登录

Linux一般作为服务器使用，而服务器一般放在机房，你不可能在机房操作你的Linux服务器。

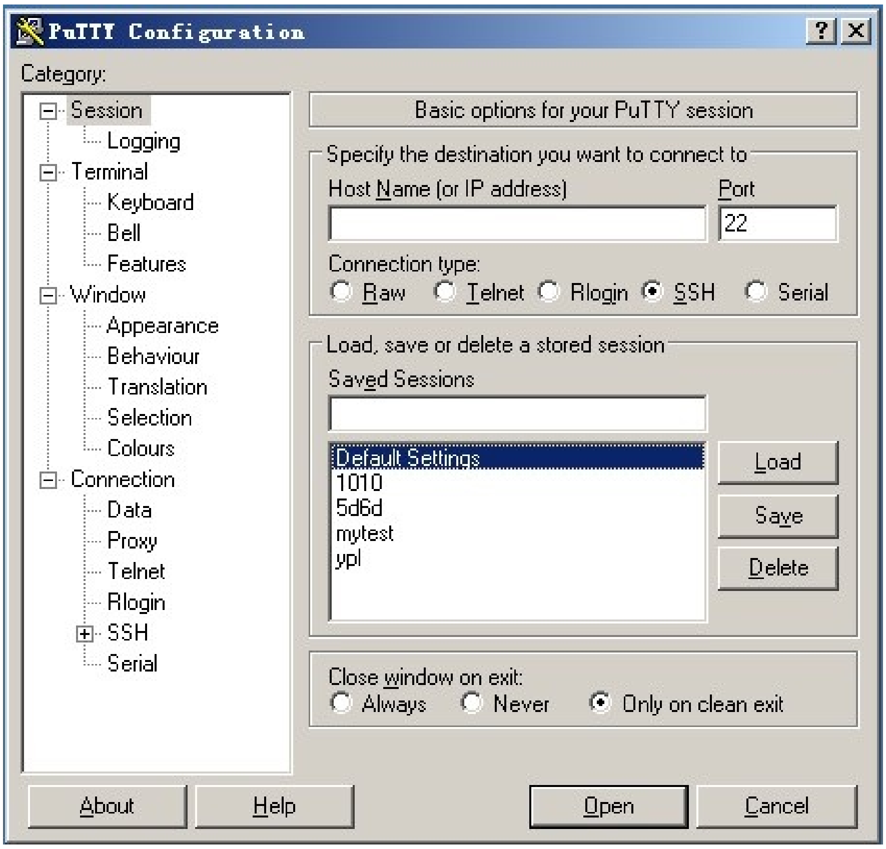
这事我们就需要远程登录到Linux服务器来管理维护系统。

Linux系统中是通过ssh服务实现的远程登录功能，默认ssh服务端口号为 22。

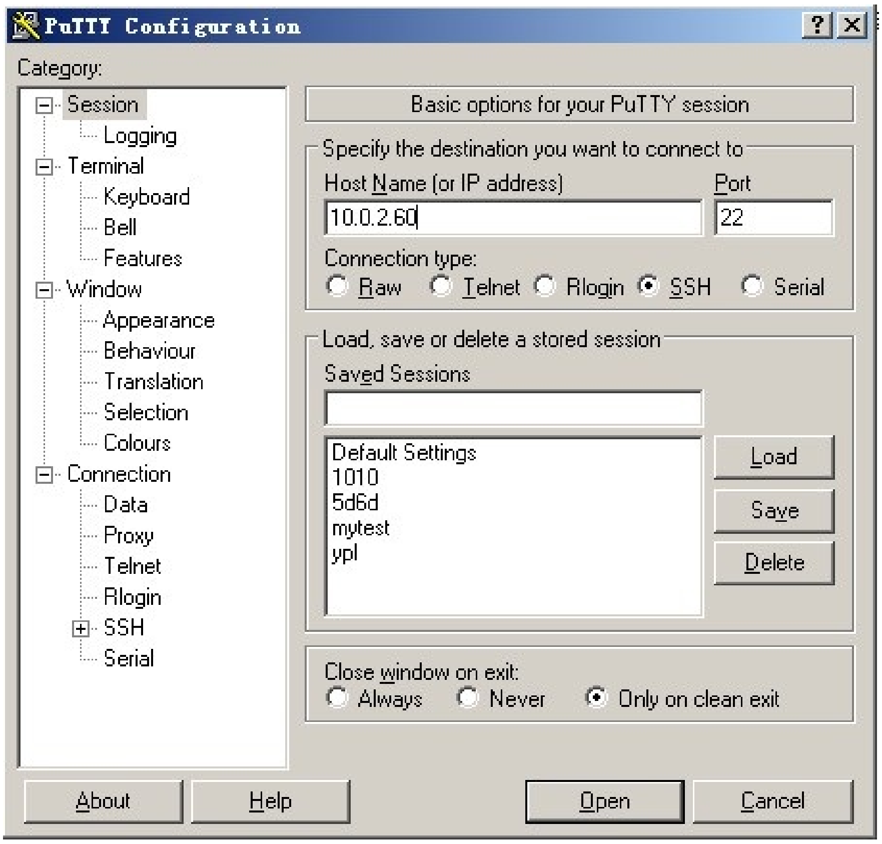
Window系统上 Linux 远程登录客户端有SecureCRT, Putty, SSH Secure Shell等，本文以Putty为例来登录远程服务器。

putty下载地址：http://www.putty.org/

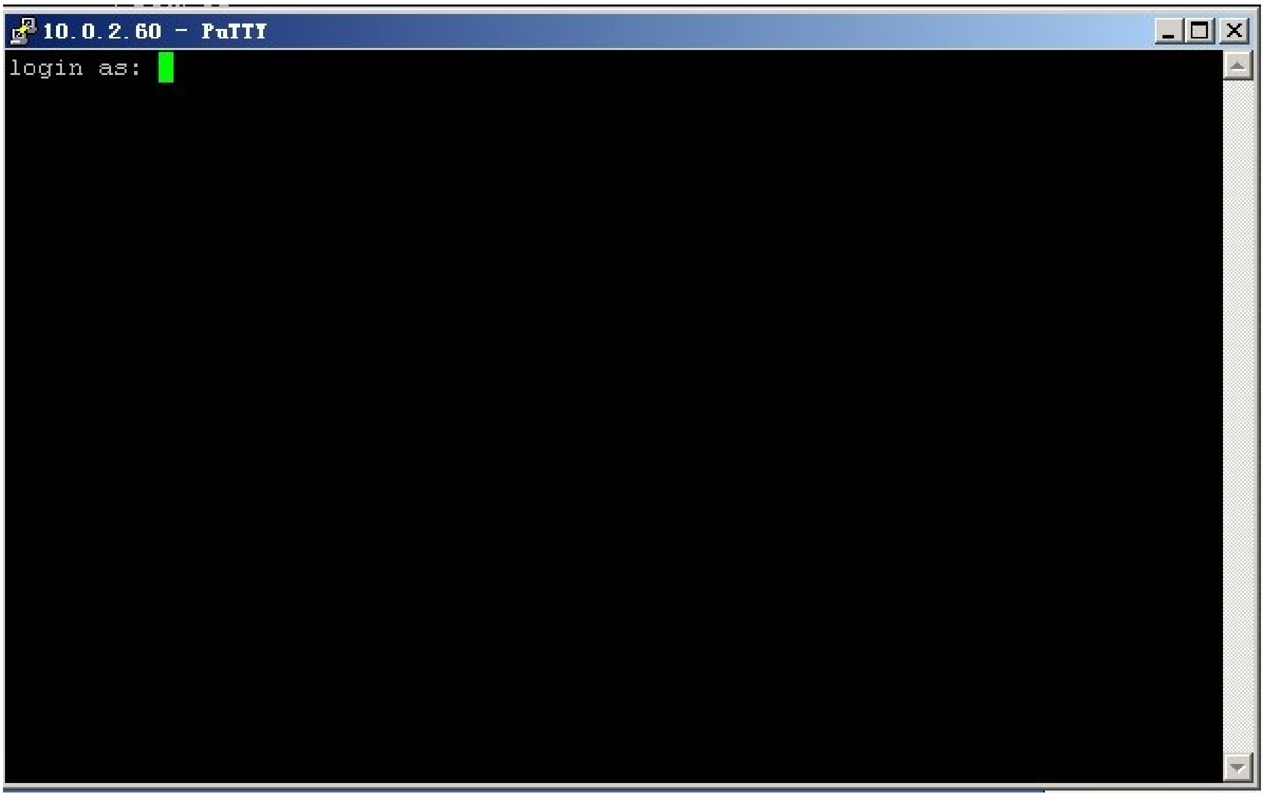
如果你下载了putty，请双击putty.exe 然后弹出如下的窗口。



在Host Name(or IP address) 下面的框中输入你要登录的远程服务器IP(可以通过ifconfig命令查看服务器ip)，然后回车。



此时，提示我们输入要登录的用户名。



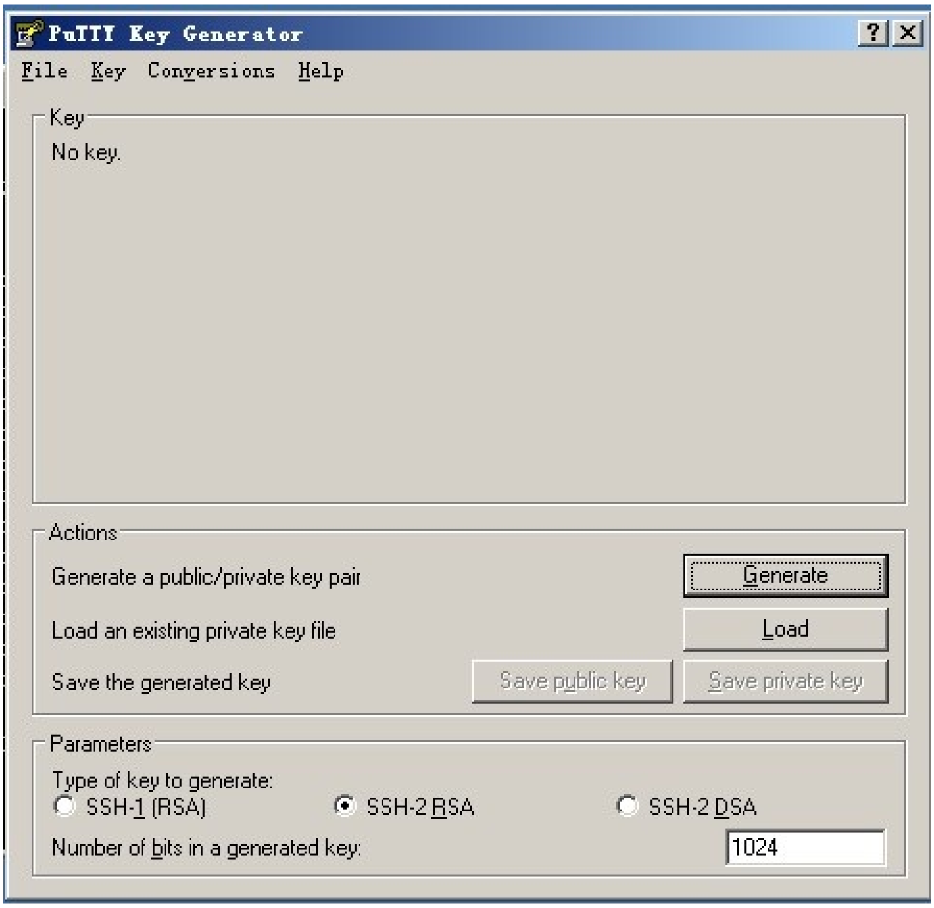
输入root 然后回车，再输入密码，就能登录到远程的linux系统了。

使用密钥认证机制远程登录linux

SSH 为 Secure Shell 的缩写，由 IETF 的网络工作小组（Network Working Group）所制定。

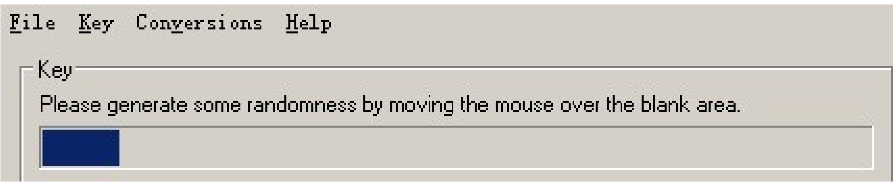
SSH 为建立在应用层和传输层基础上的安全协议。

首先使用工具 PUTTYGEN.EXE 生成密钥对。打开工具PUTTYGEN.EXE后如下图所示：

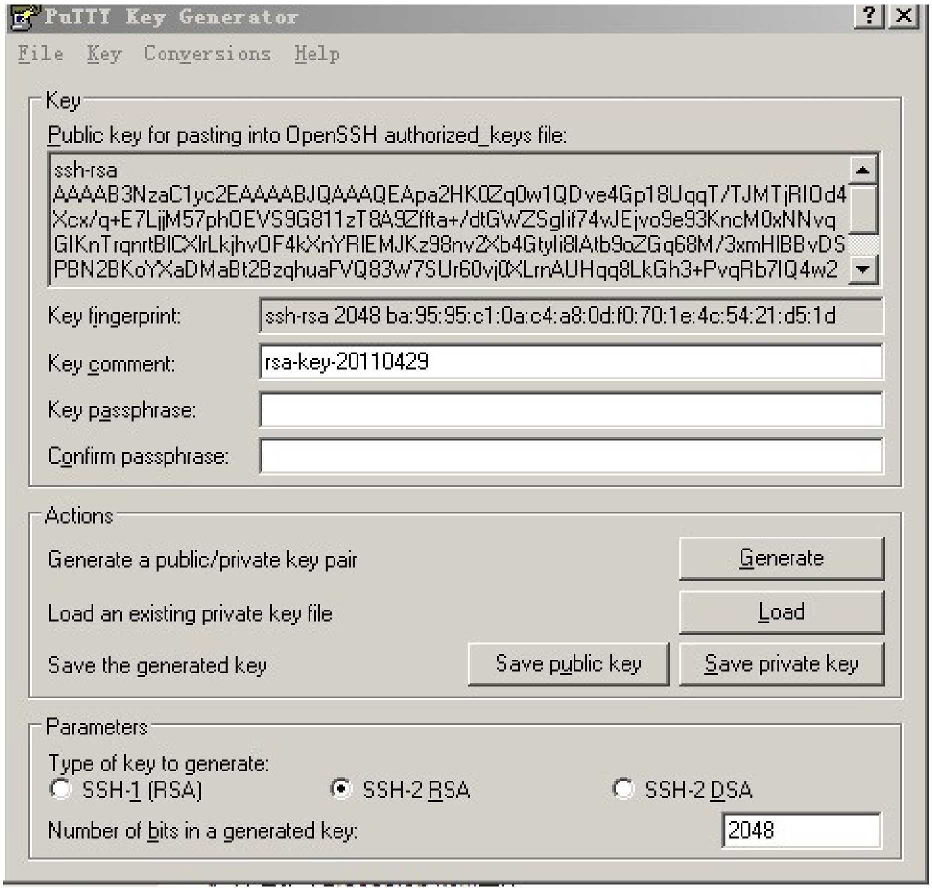


该工具可以生成三种格式的key ：SSH-1(RSA) SSH-2(RSA) SSH-2(DSA)，我们采用默认的格式即SSH-2(RSA)。Number of bits in a generated key 这个是指生成的key的大小，这个数值越大，生成的key就越复杂，安全性就越高。这里我们写2048.

然后单击Generate 开始生成密钥对：



注意的是，在这个过程中鼠标要来回的动，否则这个进度条是不会动的。



到这里，密钥对已经生成了。你可以给你的密钥输入一个密码，（在Key Passphrase那里）也可以留空。然后点 Save public key 保存公钥，点 Save private Key 保存私钥。笔者建议你放到一个比较安全的地方，一来防止别人偷窥，二来防止误删除。接下来就该到远程linux主机上设置了。

1）创建目录 /root/.ssh 并设置权限

[root@localhost ~]# mkdir /root/.ssh mkdir 命令用来创建目录，以后会详细介绍，暂时只了解即可。

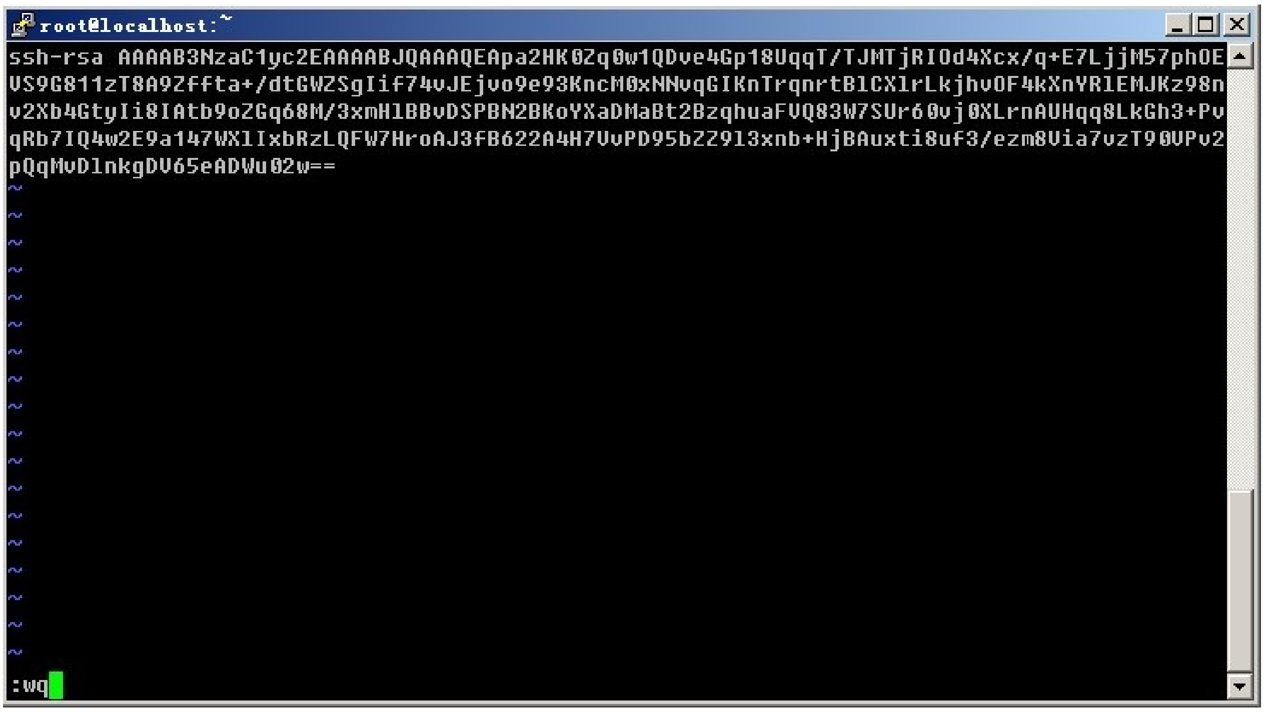
[root@localhost ~]# chmod 700 /root/.ssh chmod 命令是用来修改文件属性权限的，以后会详细介绍。

2）创建文件 / root/.ssh/authorized_keys

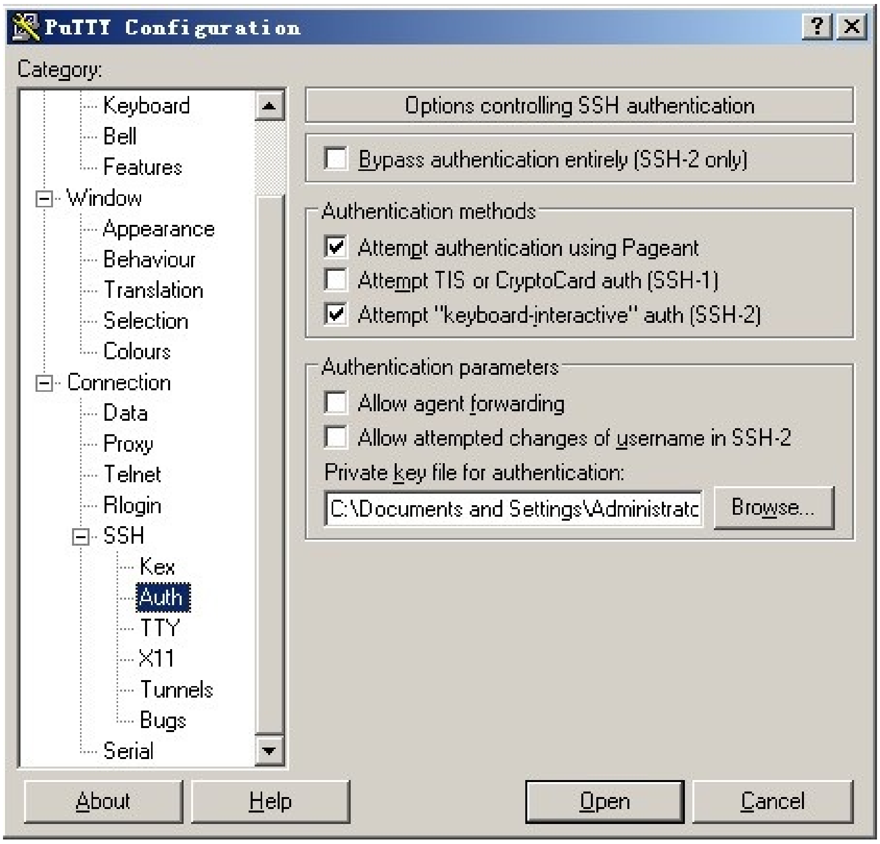
[root@localhost ~]# vim /root/.ssh/authorized_keys vim 命令是编辑一个文本文件的命令，同样在后续章节详细介绍。

3）打开刚才生成的public key 文件，建议使用写字板打开，这样看着舒服一些，复制从AAAA开头至 "---- END SSH2 PUBLIC KEY ----" 该行上的所有内容，粘贴到/root/.ssh/authorized_keys 文件中，要保证所有字符在一行。（可以先把复制的内容拷贝至记事本，然后编辑成一行载粘贴到该文件中）。

在这里要简单介绍一下，如何粘贴，用vim打开那个文件后，该文件不存在，所以vim会自动创建。按一下字母"i"然后同时按shift + Insert 进行粘贴（或者单击鼠标邮件即可），前提是已经复制到剪切板中了。粘贴好后，然后把光标移动到该行最前面输入ssh-ras， 然后按空格。再按ESC，然后输入冒号wq 即 :wq 就保存了。格式如下图：



4）再设置putty选项，点窗口左侧的SSh –> Auth， 单击窗口右侧的Browse… 选择刚刚生成的私钥， 再点Open， 此时输入root，就不用输入密码就能登录了。



如果在前面你设置了Key Passphrase， 那么此时就会提示你输入密码的。为了更加安全建议大家要设置一个Key Passphrase。

Linux 文件基本属性

Linux系统是一种典型的多用户系统，不同的用户处于不同的地位，拥有不同的权限。为了保护系统的安全性，Linux系统对不同的用户访问同一文件（包括目录文件）的权限做了不同的规定。

在Linux中我们可以使用ll或者ls –l命令来显示一个文件的属性以及文件所属的用户和组，如：

```
[root@www /]# ls -l
total 64
dr-xr-xr-x   2 root root 4096 Dec 14  2012 bin
dr-xr-xr-x   4 root root 4096 Apr 19  2012 boot
……
```

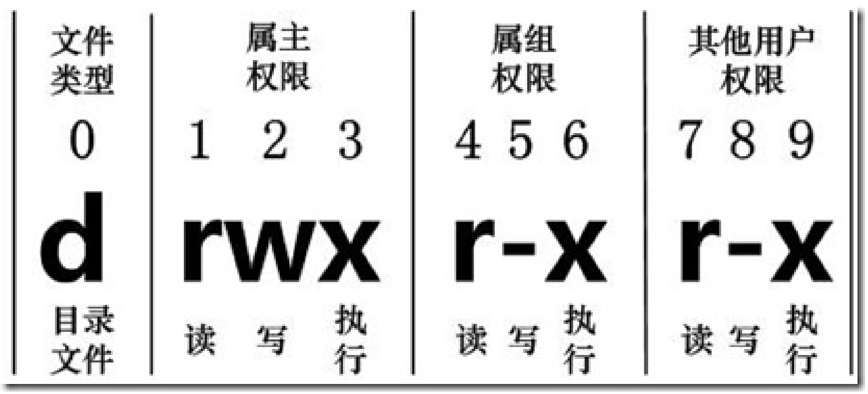
实例中，bin文件的第一个属性用"d"表示。"d"在Linux中代表该文件是一个目录文件。

在Linux中第一个字符代表这个文件是目录、文件或链接文件等等。

- 当为[*d*]则是目录
- 当为[*-*]则是文件；
- 若是[*l*]则表示为链接文档(link file)；
- 若是[*b*]则表示为装置文件里面的可供储存的接口设备(可随机存取装置)；
- 若是[*c*]则表示为装置文件里面的串行端口设备，例如键盘、鼠标(一次性读取装置)。

接下来的字符中，以三个为一组，且均为『rwx』 的三个参数的组合。其中，[r]代表可读(read)、[w]代表可写(write)、[x]代表可执行(execute)。 要注意的是，这三个权限的位置不会改变，如果没有权限，就会出现减号[-]而已。

每个文件的属性由左边第一部分的10个字符来确定（如下图）。



从左至右用0-9这些数字来表示。

第0位确定文件类型，第1-3位确定属主（该文件的所有者）拥有该文件的权限。

第4-6位确定属组（所有者的同组用户）拥有该文件的权限，第7-9位确定其他用户拥有该文件的权限。

其中，第1、4、7位表示读权限，如果用"r"字符表示，则有读权限，如果用"-"字符表示，则没有读权限；

第2、5、8位表示写权限，如果用"w"字符表示，则有写权限，如果用"-"字符表示没有写权限；第3、6、9位表示可执行权限，如果用"x"字符表示，则有执行权限，如果用"-"字符表示，则没有执行权限。

Linux文件属主和属组

```
[root@www /]# ls -l
total 64
dr-xr-xr-x   2 root root 4096 Dec 14  2012 bin
dr-xr-xr-x   4 root root 4096 Apr 19  2012 boot
……
```

对于文件来说，它都有一个特定的所有者，也就是对该文件具有所有权的用户。

同时，在Linux系统中，用户是按组分类的，一个用户属于一个或多个组。

文件所有者以外的用户又可以分为文件所有者的同组用户和其他用户。

因此，Linux系统按文件所有者、文件所有者同组用户和其他用户来规定了不同的文件访问权限。

在以上实例中，bin文件是一个目录文件，属主和属组都为root，属主有可读、可写、可执行的权限；与属主同组的其他用户有可读和可执行的权限；其他用户也有可读和可执行的权限。

更改文件属性

1、chgrp：更改文件属组

语法：

```
chgrp [-R] 属组名文件名
```

参数选项

- -R：递归更改文件属组，就是在更改某个目录文件的属组时，如果加上-R的参数，那么该目录下的所有文件的属组都会更改。

2、chown：更改文件属主，也可以同时更改文件属组

语法：

```
chown [-R] 属主名 文件名
chown [-R] 属主名：属组名 文件名
```

进入 /root 目录（~）将install.log的拥有者改为bin这个账号：

```
[root@www ~] cd ~
[root@www ~]# chown bin install.log
[root@www ~]# ls -l
-rw-r--r--  1 bin  users 68495 Jun 25 08:53 install.log
```

将install.log的拥有者与群组改回为root：

```
[root@www ~]# chown root:root install.log
[root@www ~]# ls -l
-rw-r--r--  1 root root 68495 Jun 25 08:53 install.log
```

3、chmod：更改文件9个属性

Linux文件属性有两种设置方法，一种是数字，一种是符号。

Linux文件的基本权限就有九个，分别是owner/group/others三种身份各有自己的read/write/execute权限。

先复习一下刚刚上面提到的数据：文件的权限字符为：『-rwxrwxrwx』， 这九个权限是三个三个一组的！其中，我们可以使用数字来代表各个权限，各权限的分数对照表如下：

- r:4
- w:2
- x:1

每种身份(owner/group/others)各自的三个权限(r/w/x)分数是需要累加的，例如当权限为： [-rwxrwx---] 分数则是：

- owner = rwx = 4+2+1 = 7
- group = rwx = 4+2+1 = 7
- others= --- = 0+0+0 = 0

所以等一下我们设定权限的变更时，该文件的权限数字就是770啦！变更权限的指令chmod的语法是这样的：

```
 chmod [-R] xyz 文件或目录
```

选项与参数：

- xyz : 就是刚刚提到的数字类型的权限属性，为 rwx 属性数值的相加。
- -R : 进行递归(recursive)的持续变更，亦即连同次目录下的所有文件都会变更

举例来说，如果要将.bashrc这个文件所有的权限都设定启用，那么命令如下：

```
[root@www ~]# ls -al .bashrc
-rw-r--r--  1 root root 395 Jul  4 11:45 .bashrc
[root@www ~]# chmod 777 .bashrc
[root@www ~]# ls -al .bashrc
-rwxrwxrwx  1 root root 395 Jul  4 11:45 .bashrc
```

那如果要将权限变成 *-rwxr-xr--* 呢？那么权限的分数就成为 [4+2+1][4+0+1][4+0+0]=754。

符号类型改变文件权限

还有一个改变权限的方法呦！从之前的介绍中我们可以发现，基本上就九个权限分别是(1)user (2)group (3)others三种身份啦！ 那么我们就可以藉由u, g, o来代表三种身份的权限！

此外， a 则代表 all 亦即全部的身份！那么读写的权限就可以写成r, w, x！也就是可以使用底下的方式来看：

chmod	u g o a	+(加入) -(除去) =(设定)	r w x	文件或目录

如果我们需要将文件权限设置为 -rwxr-xr-- ， 可以使用 *chmod u=rwx,g=rx,o=r* 文件名 来设定:

```
[root@www ~]# ls -al .bashrc
-rwxr-xr-x  1 root root 395 Jul  4 11:45 .bashrc
[root@www ~]# chmod  a+w  .bashrc
[root@www ~]# ls -al .bashrc
-rwxrwxrwx  1 root root 395 Jul  4 11:45 .bashrc
```

而如果是要将权限去掉而不改变其他已存在的权限呢？例如要拿掉全部人的可执行权限，则：

```
[root@www ~]# chmod  a-x  .bashrc
[root@www ~]# ls -al .bashrc
-rw-rw-rw-  1 root root 395 Jul  4 11:45 .bashrc
```

Linux 文件与目录管理

我们知道Linux的目录结构为树状结构，最顶级的目录为根目录 /。

其他目录通过挂载可以将它们添加到树中，通过解除挂载可以移除它们。

在开始本教程前我们需要先知道什么是绝对路径与相对路径。

- 绝对路径：
  路径的写法，由根目录 / 写起，例如： /usr/share/doc 这个目录。
- 相对路径：
  路径的写法，不是由 / 写起，例如由 /usr/share/doc 要到 /usr/share/man 底下时，可以写成： cd ../man 这就是相对路径的写法啦！

处理目录的常用命令

接下来我们就来看几个常见的处理目录的命令吧：

- ls: 列出目录
- cd：切换目录
- pwd：显示目前的目录
- mkdir：创建一个新的目录
- rmdir：删除一个空的目录
- cp: 复制文件或目录
- rm: 移除文件或目录

你可以使用 *man [命令]* 来查看各个命令的使用文档，如 ：man cp。

ls (列出目录)

在Linux系统当中， ls 命令可能是最常被运行的。

语法：

```
[root@www ~]# ls [-aAdfFhilnrRSt] 目录名称
[root@www ~]# ls [--color={never,auto,always}] 目录名称
[root@www ~]# ls [--full-time] 目录名称
```

选项与参数：

- -a ：全部的文件，连同隐藏档(开头为 . 的文件) 一起列出来(常用)
- -d ：仅列出目录本身，而不是列出目录内的文件数据(常用)

- -l ：长数据串列出，包含文件的属性与权限等等数据；(常用)

将家目录下的所有文件列出来(含属性与隐藏档)

```
[root@www ~]# ls -al ~
```

cd (切换目录)

cd是Change Directory的缩写，这是用来变换工作目录的命令。

语法：

```
 cd [相对路径或绝对路径]
```

```
#使用 mkdir 命令创建w3cschool.cc目录
[root@www ~]# mkdir w3cschool.cc

#使用绝对路径切换到w3cschool.cc目录
[root@www ~]# cd /root/w3cschool.cc/

#使用相对路径切换到w3cschool.cc目录
[root@www ~]# cd ./w3cschool.cc/

# 表示回到自己的家目录，亦即是 /root 这个目录
[root@www w3cschool.cc]# cd ~

# 表示去到目前的上一级目录，亦即是 /root 的上一级目录的意思；
[root@www ~]# cd ..
```

接下来大家多操作几次应该就可以很好的理解 cd 命令的。

pwd (显示目前所在的目录)

pwd是Print Working Directory的缩写，也就是显示目前所在目录的命令。

```
[root@www ~]# pwd [-P]
选项与参数：
-P  ：显示出确实的路径，而非使用连结 (link) 路径。

范例：单纯显示出目前的工作目录：
[root@www ~]# pwd
/root   <== 显示出目录啦~

范例：显示出实际的工作目录，而非连结档本身的目录名而已
[root@www ~]# cd /var/mail   <==注意，/var/mail是一个连结档
[root@www mail]# pwd
/var/mail         <==列出目前的工作目录
[root@www mail]# pwd -P
/var/spool/mail   <==怎么回事？有没有加 -P 差很多~
[root@www mail]# ls -ld /var/mail
lrwxrwxrwx 1 root root 10 Sep  4 17:54 /var/mail -> spool/mail
# 看到这里应该知道为啥了吧？因为 /var/mail 是连结档，连结到 /var/spool/mail
# 所以，加上 pwd -P 的选项后，会不以连结档的数据显示，而是显示正确的完整路径啊！
```

mkdir (创建新目录)

如果想要创建新的目录的话，那么就使用mkdir (make directory)吧。

语法：

```
mkdir [-mp] 目录名称
```

选项与参数：

- -m ：配置文件的权限喔！直接配置，不需要看默认权限 (umask) 的脸色～
- -p ：帮助你直接将所需要的目录(包含上一级目录)递回创建起来！

范例：请到/tmp底下尝试创建数个新目录看看：

```
[root@www ~]# cd /tmp
[root@www tmp]# mkdir test    <==创建一名为 test 的新目录
[root@www tmp]# mkdir test1/test2/test3/test4
mkdir: cannot create directory `test1/test2/test3/test4':
No such file or directory       <== 没办法直接创建此目录啊！
[root@www tmp]# mkdir -p test1/test2/test3/test4
```

加了这个 -p 的选项，可以自行帮你创建多层目录！

范例：创建权限为rwx--x--x的目录

```
[root@www tmp]# mkdir -m 711 test2
[root@www tmp]# ls -l
drwxr-xr-x  3 root  root 4096 Jul 18 12:50 test
drwxr-xr-x  3 root  root 4096 Jul 18 12:53 test1
drwx--x--x  2 root  root 4096 Jul 18 12:54 test2
```

上面的权限部分，如果没有加上 -m 来强制配置属性，系统会使用默认属性。

如果我们使用 -m ，如上例我们给予 -m 711 来给予新的目录 drwx--x--x 的权限。

rmdir (删除空的目录)

语法：

```
 rmdir [-p] 目录名称
```

选项与参数：

- **-p** ：连同上一级『空的』目录也一起删除

删除 w3cschool.cc 目录

```
[root@www tmp]# rmdir w3cschool.cc/
```

范例：将於mkdir范例中创建的目录(/tmp底下)删除掉！

```
[root@www tmp]# ls -l   <==看看有多少目录存在？
drwxr-xr-x  3 root  root 4096 Jul 18 12:50 test
drwxr-xr-x  3 root  root 4096 Jul 18 12:53 test1
drwx--x--x  2 root  root 4096 Jul 18 12:54 test2
[root@www tmp]# rmdir test   <==可直接删除掉，没问题
[root@www tmp]# rmdir test1  <==因为尚有内容，所以无法删除！
rmdir: `test1': Directory not empty
[root@www tmp]# rmdir -p test1/test2/test3/test4
[root@www tmp]# ls -l        <==您看看，底下的输出中test与test1不见了！
drwx--x--x  2 root  root 4096 Jul 18 12:54 test2
```

利用 -p 这个选项，立刻就可以将 test1/test2/test3/test4 一次删除。

不过要注意的是，这个 rmdir 仅能删除空的目录，你可以使用 rm 命令来删除非空目录。

cp (复制文件或目录)

cp 即拷贝文件和目录。

语法:

```
[root@www ~]# cp [-adfilprsu] 来源档(source) 目标档(destination)
[root@www ~]# cp [options] source1 source2 source3 .... directory
```

选项与参数：

- -a ：相当於 -pdr 的意思，至於 pdr 请参考下列说明；(常用) *-d* ：若来源档为连结档的属性*(link file)*，则复制连结档属性而非文件本身； -f ：为强制(force)的意思，若目标文件已经存在且无法开启，则移除后再尝试一次； *-i* ：若目标档*(destination)*已经存在时，在覆盖时会先询问动作的进行*(*常用*)* -l ：进行硬式连结(hard link)的连结档创建，而非复制文件本身； *-p* ：连同文件的属性一起复制过去，而非使用默认属性*(*备份常用*)*； -r ：递回持续复制，用於目录的复制行为；(常用) *-s* ：复制成为符号连结档 *(symbolic link)*，亦即『捷径』文件； -u ：若 destination 比 source 旧才升级 destination ！

用root身份，将家目录下的 .bashrc 复制到 /tmp 下，并更名为 bashr

```
[root@www ~]# cp ~/.bashrc /tmp/bashrc
[root@www ~]# cp -i ~/.bashrc /tmp/bashrc
cp: overwrite `/tmp/bashrc'? n  <==n不覆盖，y为覆盖
```

rm (移除文件或目录)

语法：

```
rm [-fir] 文件或目录
```

选项与参数：

- -f ：就是 force 的意思，忽略不存在的文件，不会出现警告信息；
- -i ：互动模式，在删除前会询问使用者是否动作
- -r ：递回删除啊！最常用在目录的删除了！这是非常危险的选项！！！

将刚刚在 cp 的范例中创建的 bashrc 删除掉！

```
[root@www tmp]# rm -i bashrc
rm: remove regular file `bashrc'? y
```

如果加上 -i 的选项就会主动询问喔，避免你删除到错误的档名！

mv (移动文件与目录，或修改名称)

语法：

```
[root@www ~]# mv [-fiu] source destination
[root@www ~]# mv [options] source1 source2 source3 .... directory
```

选项与参数：

- -f ：force 强制的意思，如果目标文件已经存在，不会询问而直接覆盖；
- -i ：若目标文件 (destination) 已经存在时，就会询问是否覆盖！
- -u ：若目标文件已经存在，且 source 比较新，才会升级 (update)

复制一文件，创建一目录，将文件移动到目录中

```
[root@www ~]# cd /tmp
[root@www tmp]# cp ~/.bashrc bashrc
[root@www tmp]# mkdir mvtest
[root@www tmp]# mv bashrc mvtest
```

将某个文件移动到某个目录去，就是这样做！

将刚刚的目录名称更名为 mvtest2

```
[root@www tmp]# mv mvtest mvtest2
```

Linux 文件内容查看

Linux系统中使用以下命令来查看文件的内容：

- cat 由第一行开始显示文件内容
- tac 从最后一行开始显示，可以看出 tac 是 cat 的倒著写！
- nl 显示的时候，顺道输出行号！
- more 一页一页的显示文件内容
- less 与 more 类似，但是比 more 更好的是，他可以往前翻页！
- head 只看头几行
- tail 只看尾巴几行

你可以使用 *man [命令]*来查看各个命令的使用文档，如 ：man cp。

cat

由第一行开始显示文件内容

语法：

```
cat [-AbEnTv]
```

选项与参数：

- -A ：相当於 -vET 的整合选项，可列出一些特殊字符而不是空白而已；
- -b ：列出行号，仅针对非空白行做行号显示，空白行不标行号！
- -E ：将结尾的断行字节 $ 显示出来；
- -n ：列印出行号，连同空白行也会有行号，与 -b 的选项不同；
- -T ：将 [tab] 按键以 ^I 显示出来；
- -v ：列出一些看不出来的特殊字符

检看 /etc/issue 这个文件的内容：

```
[root@www ~]# cat /etc/issue
CentOS release 6.4 (Final)
Kernel \r on an \m
```

tac

tac与cat命令刚好相反，文件内容从最后一行开始显示，可以看出 tac 是 cat 的倒着写！如：

```
[root@www ~]# tac /etc/issue

Kernel \r on an \m
CentOS release 6.4 (Final)
```

nl

显示行号

语法：

```
nl [-bnw] 文件
```

选项与参数：

- -b ：指定行号指定的方式，主要有两种：
  -b a ：表示不论是否为空行，也同样列出行号(类似 cat -n)；
  -b t ：如果有空行，空的那一行不要列出行号(默认值)；
- -n ：列出行号表示的方法，主要有三种：
  -n ln ：行号在萤幕的最左方显示；
  -n rn ：行号在自己栏位的最右方显示，且不加 0 ；
  -n rz ：行号在自己栏位的最右方显示，且加 0 ；
- -w ：行号栏位的占用的位数。

范例一：用 nl 列出 /etc/issue 的内容

```
[root@www ~]# nl /etc/issue
     1  CentOS release 6.4 (Final)
     2  Kernel \r on an \m
```

more

一页一页翻动

```
[root@www ~]# more /etc/man.config
#
# Generated automatically from man.conf.in by the
# configure script.
#
# man.conf from man-1.6d
....(中间省略)....
--More--(28%)  <== 重点在这一行喔！你的光标也会在这里等待你的命令
```

在 more 这个程序的运行过程中，你有几个按键可以按的：

- 空白键 (space)：代表向下翻一页；
- Enter ：代表向下翻『一行』；
- /字串 ：代表在这个显示的内容当中，向下搜寻『字串』这个关键字；
- :f ：立刻显示出档名以及目前显示的行数；
- q ：代表立刻离开 more ，不再显示该文件内容。
- b 或 [ctrl]-b ：代表往回翻页，不过这动作只对文件有用，对管线无用。

less

一页一页翻动，以下实例输出/etc/man.config文件的内容：

```
[root@www ~]# less /etc/man.config
#
# Generated automatically from man.conf.in by the
# configure script.
#
# man.conf from man-1.6d
....(中间省略)....
:   <== 这里可以等待你输入命令！
```

less运行时可以输入的命令有：

- 空白键 ：向下翻动一页；
- [pagedown]：向下翻动一页；
- [pageup] ：向上翻动一页；
- /字串 ：向下搜寻『字串』的功能；
- ?字串 ：向上搜寻『字串』的功能；
- n ：重复前一个搜寻 (与 / 或 ? 有关！)
- N ：反向的重复前一个搜寻 (与 / 或 ? 有关！)
- q ：离开 less 这个程序；

head

取出文件前面几行

语法：

```
head [-n number] 文件
```

选项与参数：

- -n ：后面接数字，代表显示几行的意思

```
[root@www ~]# head /etc/man.config
```

默认的情况中，显示前面 10 行！若要显示前 20 行，就得要这样：

```
[root@www ~]# head -n 20 /etc/man.config
```

tail

取出文件后面几行

语法：

```
tail [-n number] 文件
```

选项与参数：

- -n ：后面接数字，代表显示几行的意思
- -f ：表示持续侦测后面所接的档名，要等到按下[ctrl]-c才会结束tail的侦测

```
[root@www ~]# tail /etc/man.config
# 默认的情况中，显示最后的十行！若要显示最后的 20 行，就得要这样：
[root@www ~]# tail -n 20 /etc/man.config
```

Linux 用户和用户组管理

Linux系统是一个多用户多任务的分时操作系统，任何一个要使用系统资源的用户，都必须首先向系统管理员申请一个账号，然后以这个账号的身份进入系统。

用户的账号一方面可以帮助系统管理员对使用系统的用户进行跟踪，并控制他们对系统资源的访问；另一方面也可以帮助用户组织文件，并为用户提供安全性保护。

每个用户账号都拥有一个惟一的用户名和各自的口令。

用户在登录时键入正确的用户名和口令后，就能够进入系统和自己的主目录。

实现用户账号的管理，要完成的工作主要有如下几个方面：

- 用户账号的添加、删除与修改。
- 用户口令的管理。
- 用户组的管理。

一、Linux系统用户账号的管理

用户账号的管理工作主要涉及到用户账号的添加、修改和删除。

添加用户账号就是在系统中创建一个新账号，然后为新账号分配用户号、用户组、主目录和登录Shell等资源。刚添加的账号是被锁定的，无法使用。

1、添加新的用户账号使用useradd命令，其语法如下：

```
useradd 选项 用户名
```

参数说明：

- 选项:
    - -c comment 指定一段注释性描述。
    - -d 目录 指定用户主目录，如果此目录不存在，则同时使用-m选项，可以创建主目录。
    - -g 用户组 指定用户所属的用户组。
    - -G 用户组，用户组 指定用户所属的附加组。
    - -s Shell文件 指定用户的登录Shell。
    - -u 用户号 指定用户的用户号，如果同时有-o选项，则可以重复使用其他用户的标识号。
- 用户名:

指定新账号的登录名。

实例1

```
# useradd –d /usr/sam -m sam
```

此命令创建了一个用户sam，其中-d和-m选项用来为登录名sam产生一个主目录/usr/sam（/usr为默认的用户主目录所在的父目录）。

实例2

```
# useradd -s /bin/sh -g group –G adm,root gem
```

此命令新建了一个用户gem，该用户的登录Shell是 `/bin/sh`， 它属于group用户组，同时又属于adm和root用户组，其中group用户组是其主组。

这里可能新建组：`#groupadd group及groupadd adm`

增加用户账号就是在/etc/passwd文件中为新用户增加一条记录，同时更新其他系统文件如/etc/shadow, /etc/group等。

Linux提供了集成的系统管理工具userconf，它可以用来对用户账号进行统一管理。

3、删除帐号

如果一个用户的账号不再使用，可以从系统中删除。删除用户账号就是要将/etc/passwd等系统文件中的该用户记录删除，必要时还删除用户的主目录。

删除一个已有的用户账号使用`userdel`命令，其格式如下：

```
userdel 选项 用户名
```

常用的选项是-r，它的作用是把用户的主目录一起删除。

例如：

```
# userdel sam
```

此命令删除用户sam在系统文件中（主要是/etc/passwd, /etc/shadow, /etc/group等）的记录，同时删除用户的主目录。

4、修改帐号

修改用户账号就是根据实际情况更改用户的有关属性，如用户号、主目录、用户组、登录Shell等。

修改已有用户的信息使用 `usermod` 命令，其格式如下：

```
usermod 选项 用户名
```

常用的选项包括 `-c, -d, -m, -g, -G, -s, -u以及-o等` ，这些选项的意义与 `useradd` 命令中的选项一样，可以为用户指定新的资源值。

另外，有些系统可以使用选项：-l 新用户名

这个选项指定一个新的账号，即将原来的用户名改为新的用户名。

例如：

```
# usermod -s /bin/ksh -d /home/z –g developer sam
```

此命令将用户sam的登录Shell修改为ksh，主目录改为/home/z，用户组改为developer。

5、用户口令的管理

用户管理的一项重要内容是用户口令的管理。用户账号刚创建时没有口令，但是被系统锁定，无法使用，必须为其指定口令后才可以使用，即使是指定空口令。

指定和修改用户口令的Shell命令是 `passwd` 。超级用户可以为自己和其他用户指定口令，普通用户只能用它修改自己的口令。命令的格式为：

```
passwd 选项 用户名
```

可使用的选项：

- -l 锁定口令，即禁用账号。
- -u 口令解锁。
- -d 使账号无口令。
- -f 强迫用户下次登录时修改口令。

如果默认用户名，则修改当前用户的口令。

例如，假设当前用户是sam，则下面的命令修改该用户自己的口令：

```
$ passwd
Old password:******
New password:*******
Re-enter new password:*******
```

如果是超级用户，可以用下列形式指定任何用户的口令：

```
# passwd sam
New password:*******
Re-enter new password:*******
```

普通用户修改自己的口令时，passwd命令会先询问原口令，验证后再要求用户输入两遍新口令，如果两次输入的口令一致，则将这个口令指定给用户；而超级用户为用户指定口令时，就不需要知道原口令。

为了系统安全起见，用户应该选择比较复杂的口令，例如最好使用8位长的口令，口令中包含有大写、小写字母和数字，并且应该与姓名、生日等不相同。

为用户指定空口令时，执行下列形式的命令：

```
# passwd -d sam
```

此命令将用户sam的口令删除，这样用户sam下一次登录时，系统就不再询问口令。

passwd命令还可以用-l(lock)选项锁定某一用户，使其不能登录，例如：

```
# passwd -l sam
```

二、Linux系统用户组的管理

每个用户都有一个用户组，系统可以对一个用户组中的所有用户进行集中管理。不同Linux 系统对用户组的规定有所不同，如Linux下的用户属于与它同名的用户组，这个用户组在创建用户时同时创建。

用户组的管理涉及用户组的添加、删除和修改。组的增加、删除和修改实际上就是对/etc/group文件的更新。

1、增加一个新的用户组使用groupadd命令。其格式如下：

```
groupadd 选项 用户组
```

可以使用的选项有：

- -g GID 指定新用户组的组标识号（GID）。
- -o 一般与-g选项同时使用，表示新用户组的GID可以与系统已有用户组的GID相同。

实例1：

```
# groupadd group1
```

此命令向系统中增加了一个新组group1，新组的组标识号是在当前已有的最大组标识号的基础上加1。

实例**2**：

```
# groupadd -g 101 group2
```

此命令向系统中增加了一个新组group2，同时指定新组的组标识号是101。

2、如果要删除一个已有的用户组，使用**groupdel**命令，其格式如下：

```
groupdel 用户组
```

例如：

```
# groupdel group1
```

此命令从系统中删除组group1。

3、修改用户组的属性使用**groupmod**命令。其语法如下：

```
groupmod 选项 用户组
```

常用的选项有：

- -g GID 为用户组指定新的组标识号。
- -o 与-g选项同时使用，用户组的新GID可以与系统已有用户组的GID相同。
- -n新用户组 将用户组的名字改为新名字

实例**1**：

```
# groupmod -g 102 group2
```

此命令将组group2的组标识号修改为102。

实例**2**：

```
# groupmod –g 10000 -n group3 group2
```

此命令将组group2的标识号改为10000，组名修改为group3。

4、如果一个用户同时属于多个用户组，那么用户可以在用户组之间切换，以便具有其他用户组的权限。

用户可以在登录后，使用命令newgrp切换到其他用户组，这个命令的参数就是目的用户组。例如：

```
$ newgrp root
```

这条命令将当前用户切换到root用户组，前提条件是root用户组确实是该用户的主组或附加组。类似于用户账号的管理，用户组的管理也可以通过集成的系统管理工具来完成。

三、与用户账号有关的系统文件

完成用户管理的工作有许多种方法，但是每一种方法实际上都是对有关的系统文件进行修改。

与用户和用户组相关的信息都存放在一些系统文件中，这些文件包括/etc/passwd, /etc/shadow, /etc/group等。

下面分别介绍这些文件的内容。

1、/etc/passwd文件是用户管理工作涉及的最重要的一个文件。

Linux系统中的每个用户都在/etc/passwd文件中有一个对应的记录行，它记录了这个用户的一些基本属性。

这个文件对所有用户都是可读的。它的内容类似下面的例子：

```
＃ cat /etc/passwd

root:x:0:0:Superuser:/:
daemon:x:1:1:System daemons:/etc:
bin:x:2:2:Owner of system commands:/bin:
sys:x:3:3:Owner of system files:/usr/sys:
adm:x:4:4:System accounting:/usr/adm:
uucp:x:5:5:UUCP administrator:/usr/lib/uucp:
auth:x:7:21:Authentication administrator:/tcb/files/auth:
cron:x:9:16:Cron daemon:/usr/spool/cron:
listen:x:37:4:Network daemon:/usr/net/nls:
lp:x:71:18:Printer administrator:/usr/spool/lp:
sam:x:200:50:Sam san:/usr/sam:/bin/sh
```

从上面的例子我们可以看到，/etc/passwd中一行记录对应着一个用户，每行记录又被冒号(:)分隔为7个字段，其格式和具体含义如下：

```
用户名:口令:用户标识号:组标识号:注释性描述:主目录:登录Shell
```

1）"用户名"是代表用户账号的字符串。

通常长度不超过8个字符，并且由大小写字母和/或数字组成。登录名中不能有冒号(:)，因为冒号在这里是分隔符。

为了兼容起见，登录名中最好不要包含点字符(.)，并且不使用连字符(-)和加号(+)打头。

2）“口令”一些系统中，存放着加密后的用户口令字。

虽然这个字段存放的只是用户口令的加密串，不是明文，但是由于/etc/passwd文件对所有用户都可读，所以这仍是一个安全隐患。因此，现在许多Linux 系统（如SVR4）都使用了shadow技术，把真正的加密后的用户口令字存放到/etc/shadow文件中，而在/etc/passwd文件的口令字段中只存放一个特殊的字符，例如“x”或者“*”。

3）“用户标识号”是一个整数，系统内部用它来标识用户。

一般情况下它与用户名是一一对应的。如果几个用户名对应的用户标识号是一样的，系统内部将把它们视为同一个用户，但是它们可以有不同的口令、不同的主目录以及不同的登录Shell等。

通常用户标识号的取值范围是0～65 535。0是超级用户root的标识号，1～99由系统保留，作为管理账号，普通用户的标识号从100开始。在Linux系统中，这个界限是500。

4）“组标识号”字段记录的是用户所属的用户组。

它对应着/etc/group文件中的一条记录。

5)“注释性描述”字段记录着用户的一些个人情况。

例如用户的真实姓名、电话、地址等，这个字段并没有什么实际的用途。在不同的Linux 系统中，这个字段的格式并没有统一。在许多Linux系统中，这个字段存放的是一段任意的注释性描述文字，用做finger命令的输出。

6)“主目录”，也就是用户的起始工作目录。

它是用户在登录到系统之后所处的目录。在大多数系统中，各用户的主目录都被组织在同一个特定的目录下，而用户主目录的名称就是该用户的登录名。各用户对自己的主目录有读、写、执行（搜索）权限，其他用户对此目录的访问权限则根据具体情况设置。

7)用户登录后，要启动一个进程，负责将用户的操作传给内核，这个进程是用户登录到系统后运行的命令解释器或某个特定的程序，即Shell。

Shell是用户与Linux系统之间的接口。Linux的Shell有许多种，每种都有不同的特点。常用的有sh(Bourne Shell), csh(C Shell), ksh(Korn Shell), tcsh(TENEX/TOPS-20 type C Shell), bash(Bourne Again Shell)等。

系统管理员可以根据系统情况和用户习惯为用户指定某个Shell。如果不指定Shell，那么系统使用sh为默认的登录Shell，即这个字段的值为/bin/sh。

用户的登录Shell也可以指定为某个特定的程序（此程序不是一个命令解释器）。

利用这一特点，我们可以限制用户只能运行指定的应用程序，在该应用程序运行结束后，用户就自动退出了系统。有些Linux 系统要求只有那些在系统中登记了的程序才能出现在这个字段中。

8)系统中有一类用户称为伪用户（psuedo users）。

这些用户在/etc/passwd文件中也占有一条记录，但是不能登录，因为它们的登录Shell为空。它们的存在主要是方便系统管理，满足相应的系统进程对文件属主的要求。

常见的伪用户如下所示：

```
伪 用 户 含 义
bin 拥有可执行的用户命令文件
sys 拥有系统文件
adm 拥有帐户文件
uucp UUCP使用
lp lp或lpd子系统使用
nobody NFS使用
```

拥有帐户文件

1、除了上面列出的伪用户外，还有许多标准的伪用户，例如：audit, cron, mail, usenet等，它们也都各自为相关的进程和文件所需要。

由于/etc/passwd文件是所有用户都可读的，如果用户的密码太简单或规律比较明显的话，一台普通的计算机就能够很容易地将它破解，因此对安全性要求较高的Linux系统都把加密后的口令字分离出来，单独存放在一个文件中，这个文件是/etc/shadow文件。 有超级用户才拥有该文件读权限，这就保证了用户密码的安全性。

2、/etc/shadow中的记录行与/etc/passwd中的一一对应，它由pwconv命令根据/etc/passwd中的数据自动产生

它的文件格式与/etc/passwd类似，由若干个字段组成，字段之间用":"隔开。这些字段是：

```
登录名:加密口令:最后一次修改时间:最小时间间隔:最大时间间隔:警告时间:不活动时间:失效时间:标志
```

1. "登录名"是与/etc/passwd文件中的登录名相一致的用户账号
2. "口令"字段存放的是加密后的用户口令字，长度为13个字符。如果为空，则对应用户没有口令，登录时不需要口令；如果含有不属于集合 { ./0-9A-Za-z }中的字符，则对应的用户不能登录。
3. "最后一次修改时间"表示的是从某个时刻起，到用户最后一次修改口令时的天数。时间起点对不同的系统可能不一样。例如在SCO Linux 中，这个时间起点是1970年1月1日。
4. "最小时间间隔"指的是两次修改口令之间所需的最小天数。
5. "最大时间间隔"指的是口令保持有效的最大天数。
6. "警告时间"字段表示的是从系统开始警告用户到用户密码正式失效之间的天数。
7. "不活动时间"表示的是用户没有登录活动但账号仍能保持有效的最大天数。
8. "失效时间"字段给出的是一个绝对的天数，如果使用了这个字段，那么就给出相应账号的生存期。期满后，该账号就不再是一个合法的账号，也就不能再用来登录了。

下面是/etc/shadow的一个例子：

```
# cat /etc/shadow

root:Dnakfw28zf38w:8764:0:168:7:::
daemon:*::0:0::::
bin:*::0:0::::
sys:*::0:0::::
adm:*::0:0::::
uucp:*::0:0::::
nuucp:*::0:0::::
auth:*::0:0::::
cron:*::0:0::::
listen:*::0:0::::
lp:*::0:0::::
sam:EkdiSECLWPdSa:9740:0:0::::
```

3、用户组的所有信息都存放在/etc/group文件中。

将用户分组是Linux 系统中对用户进行管理及控制访问权限的一种手段。

每个用户都属于某个用户组；一个组中可以有多个用户，一个用户也可以属于不同的组。

当一个用户同时是多个组中的成员时，在/etc/passwd文件中记录的是用户所属的主组，也就是登录时所属的默认组，而其他组称为附加组。

用户要访问属于附加组的文件时，必须首先使用newgrp命令使自己成为所要访问的组中的成员。

用户组的所有信息都存放在/etc/group文件中。此文件的格式也类似于/etc/passwd文件，由冒号(:)隔开若干个字段，这些字段有：

```
组名:口令:组标识号:组内用户列表
```

1. "组名"是用户组的名称，由字母或数字构成。与/etc/passwd中的登录名一样，组名不应重复。
2. "口令"字段存放的是用户组加密后的口令字。一般Linux 系统的用户组都没有口令，即这个字段一般为空，或者是*。
3. "组标识号"与用户标识号类似，也是一个整数，被系统内部用来标识组。
4. "组内用户列表"是属于这个组的所有用户的列表/b]，不同用户之间用逗号(,)分隔。这个用户组可能是用户的主组，也可能是附加组。

/etc/group文件的一个例子如下：

```
root::0:root
bin::2:root,bin
sys::3:root,uucp
adm::4:root,adm
daemon::5:root,daemon
lp::7:root,lp
users::20:root,sam
```

四、添加量用户批

添加和删除用户对每位Linux系统管理员都是轻而易举的事，比较棘手的是如果要添加几十个、上百个甚至上千个用户时，我们不太可能还使用useradd一个一个地添加，必然要找一种简便的创建大量用户的方法。Linux系统提供了创建大量用户的工具，可以让您立即创建大量用户，方法如下：

（1）先编辑一个文本用户文件。

每一列按照 `/etc/passwd` 密码文件的格式书写，要注意每个用户的用户名、UID、宿主目录都不可以相同，其中密码栏可以留做空白或输入x号。一个范例文件user.txt内容如下：

```
user001::600:100:user:/home/user001:/bin/bash
user002::601:100:user:/home/user002:/bin/bash
user003::602:100:user:/home/user003:/bin/bash
user004::603:100:user:/home/user004:/bin/bash
user005::604:100:user:/home/user005:/bin/bash
user006::605:100:user:/home/user006:/bin/bash
```

（2）以root身份执行命令 `/usr/sbin/newusers`，从刚创建的用户文件`user.txt`中导入数据，创建用户：

```
# newusers < user.txt
```

然后可以执行命令 `vipw` 或 `vi /etc/passwd` 检查 `/etc/passwd` 文件是否已经出现这些用户的数据，并且用户的宿主目录是否已经创建。

（3）执行命令/usr/sbin/pwunconv。

将 `/etc/shadow` 产生的 `shadow` 密码解码，然后回写到 `/etc/passwd` 中，并将`/etc/shadow`的`shadow`密码栏删掉。这是为了方便下一步的密码转换工作，即先取消 `shadow password` 功能。

```
# pwunconv
```

（4）编辑每个用户的密码对照文件。

范例文件 `passwd.txt` 内容如下：

```
user001:密码
user002:密码
user003:密码
user004:密码
user005:密码
user006:密码
```

（5）以root身份执行命令 `/usr/sbin/chpasswd`。

创建用户密码，`chpasswd` 会将经过 `/usr/bin/passwd` 命令编码过的密码写入 `/etc/passwd` 的密码栏。

```
# chpasswd < passwd.txt
```

（6）确定密码经编码写入/etc/passwd的密码栏后。

执行命令 `/usr/sbin/pwconv` 将密码编码为 `shadow password`，并将结果写入 `/etc/shadow`。

```
# pwconv
```

这样就完成了大量用户的创建了，之后您可以到/home下检查这些用户宿主目录的权限设置是否都正确，并登录验证用户密码是否正确。

Linux 磁盘管理

Linux磁盘管理好坏管理直接关系到整个系统的性能问题。

Linux磁盘管理常用三个命令为df、du和fdisk。

- df：列出文件系统的整体磁盘使用量
- du：检查磁盘空间使用量
- fdisk：用于磁盘分区

df

df命令参数功能：检查文件系统的磁盘空间占用情况。可以利用该命令来获取硬盘被占用了多少空间，目前还剩下多少空间等信息。

语法：

```
df [-ahikHTm] [目录或文件名]
```

选项与参数：

- -a ：列出所有的文件系统，包括系统特有的 /proc 等文件系统；
- -k ：以 KBytes 的容量显示各文件系统；
- -m ：以 MBytes 的容量显示各文件系统；
- -h ：以人们较易阅读的 GBytes, MBytes, KBytes 等格式自行显示；
- -H ：以 M=1000K 取代 M=1024K 的进位方式；
- -T ：显示文件系统类型, 连同该 partition 的 filesystem 名称 (例如 ext3) 也列出；
- -i ：不用硬盘容量，而以 inode 的数量来显示

实例 1

将系统内所有的文件系统列出来！

```
[root@www ~]# df
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/hdc2         9920624   3823112   5585444  41% /
/dev/hdc3         4956316    141376   4559108   4% /home
/dev/hdc1          101086     11126     84741  12% /boot
tmpfs              371332         0    371332   0% /dev/shm
```

在 Linux 底下如果 df 没有加任何选项，那么默认会将系统内所有的 (不含特殊内存内的文件系统与 swap) 都以 1 Kbytes 的容量来列出来！

实例 2

将容量结果以易读的容量格式显示出来

```
[root@www ~]# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hdc2             9.5G  3.7G  5.4G  41% /
/dev/hdc3             4.8G  139M  4.4G   4% /home
/dev/hdc1              99M   11M   83M  12% /boot
tmpfs                 363M     0  363M   0% /dev/shm
```

实例 3

将系统内的所有特殊文件格式及名称都列出来

```
[root@www ~]# df -aT
Filesystem    Type 1K-blocks    Used Available Use% Mounted on
/dev/hdc2     ext3   9920624 3823112   5585444  41% /
proc          proc         0       0         0   -  /proc
sysfs        sysfs         0       0         0   -  /sys
devpts      devpts         0       0         0   -  /dev/pts
/dev/hdc3     ext3   4956316  141376   4559108   4% /home
/dev/hdc1     ext3    101086   11126     84741  12% /boot
tmpfs        tmpfs    371332       0    371332   0% /dev/shm
none   binfmt_misc         0       0         0   -  /proc/sys/fs/binfmt_misc
sunrpc  rpc_pipefs         0       0         0   -  /var/lib/nfs/rpc_pipefs
```

实例 4

将 /etc 底下的可用的磁盘容量以易读的容量格式显示

```
[root@www ~]# df -h /etc
Filesystem            Size  Used Avail Use% Mounted on
/dev/hdc2             9.5G  3.7G  5.4G  41% /
```

du

inux du命令也是查看使用空间的，但是与df命令不同的是Linux du命令是对文件和目录磁盘使用的空间的查看，还是和df命令有一些区别的，这里介绍Linux du命令。

语法：

```
du [-ahskm] 文件或目录名称
```

选项与参数：

- -a ：列出所有的文件与目录容量，因为默认仅统计目录底下的文件量而已。
- -h ：以人们较易读的容量格式 (G/M) 显示；

- -s ：列出总量而已，而不列出每个各别的目录占用容量；
- -S ：不包括子目录下的总计，与 -s 有点差别。
- -k ：以 KBytes 列出容量显示；
- -m ：以 MBytes 列出容量显示；

实例 1

列出目前目录下的所有文件容量

```
[root@www ~]# du
8       ./test4     <==每个目录都会列出来
8       ./test2
....中间省略....
12      ./.gconfd   <==包括隐藏文件的目录
220     .           <==这个目录(.)所占用的总量
```

直接输入 du 没有加任何选项时，则 du 会分析当前所在目录的文件与目录所占用的硬盘空间。

实例 2

将文件的容量也列出来

```
[root@www ~]# du -a
12      ./install.log.syslog   <==有文件的列表了
8       ./.bash_logout
8       ./test4
8       ./test2
....中间省略....
12      ./.gconfd
220     .
```

实例 3

检查根目录底下每个目录所占用的容量

```
[root@www ~]# du -sm /*
7       /bin
6       /boot
.....中间省略....
0       /proc
.....中间省略....
1       /tmp
3859    /usr     <==系统初期最大就是他了啦！
77      /var
```

通配符 * 来代表每个目录。

与 df 不一样的是，du 这个命令其实会直接到文件系统内去搜寻所有的文件数据。

fdisk

fdisk 是 Linux 的磁盘分区表操作工具。

语法：

```
fdisk [-l] 装置名称
```

选项与参数：

- -l ：输出后面接的装置所有的分区内容。若仅有 fdisk -l 时， 则系统将会把整个系统内能够搜寻到的装置的分区均列出来。

实例 1

列出所有分区信息

```
[root@AY120919111755c246621 tmp]# fdisk -l

Disk /dev/xvda: 21.5 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

    Device Boot      Start         End      Blocks   Id  System
/dev/xvda1   *           1        2550    20480000   83  Linux
/dev/xvda2            2550        2611      490496   82  Linux swap / Solaris

Disk /dev/xvdb: 21.5 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x56f40944

    Device Boot      Start         End      Blocks   Id  System
/dev/xvdb2               1        2610    20964793+  83  Linux
```

实例 2

找出你系统中的根目录所在磁盘，并查阅该硬盘内的相关信息

```
[root@www ~]# df /            <==注意：重点在找出磁盘文件名而已
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hdc2              9920624   3823168   5585388  41% /

[root@www ~]# fdisk /dev/hdc  <==仔细看，不要加上数字喔！
The number of cylinders for this disk is set to 5005.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help):     <==等待你的输入！
```

输入 m 后，就会看到底下这些命令介绍

```
Command (m for help): m   <== 输入 m 后，就会看到底下这些命令介绍
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition            <==删除一个partition
   l   list known partition types
   m   print this menu
   n   add a new partition           <==新增一个partition
   o   create a new empty DOS partition table
   p   print the partition table     <==在屏幕上显示分割表
   q   quit without saving changes   <==不储存离开fdisk程序
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit  <==将刚刚的动作写入分割表
   x   extra functionality (experts only)
```

离开 fdisk 时按下 `q`，那么所有的动作都不会生效！相反的， 按下 `w` 就是动作生效的意思。

```
Command (m for help): p  <== 这里可以输出目前磁盘的状态

Disk /dev/hdc: 41.1 GB, 41174138880 bytes        <==这个磁盘的文件名与容量
255 heads, 63 sectors/track, 5005 cylinders      <==磁头、扇区与磁柱大小
Units = cylinders of 16065 * 512 = 8225280 bytes <==每个磁柱的大小

   Device Boot      Start         End      Blocks   Id  System
/dev/hdc1   *           1          13      104391   83  Linux
/dev/hdc2              14        1288    10241437+  83  Linux
/dev/hdc3            1289        1925     5116702+  83  Linux
/dev/hdc4            1926        5005    24740100    5  Extended
/dev/hdc5            1926        2052     1020096   82  Linux swap / Solaris
# 装置文件名 启动区否 开始磁柱    结束磁柱  1K大小容量 磁盘分区槽内的系统

Command (m for help): q
```

想要不储存离开吗？按下 q 就对了！不要随便按 w 啊！

使用 `p` 可以列出目前这颗磁盘的分割表信息，这个信息的上半部在显示整体磁盘的状态。

磁盘格式化

磁盘分割完毕后自然就是要进行文件系统的格式化，格式化的命令非常的简单，使用 `mkfs`（make filesystem） 命令。

语法：

```
mkfs [-t 文件系统格式] 装置文件名
```

选项与参数：

- -t ：可以接文件系统格式，例如 ext3, ext2, vfat 等(系统有支持才会生效)

实例 1

查看 mkfs 支持的文件格式

```
[root@www ~]# mkfs[tab][tab]
mkfs         mkfs.cramfs  mkfs.ext2    mkfs.ext3    mkfs.msdos   mkfs.vfat
```

按下两个[tab]，会发现 mkfs 支持的文件格式如上所示。

实例 2

将分区 /dev/hdc6（可指定你自己的分区） 格式化为 ext3 文件系统：

```
[root@www ~]# mkfs -t ext3 /dev/hdc6
mke2fs 1.39 (29-May-2006)
Filesystem label=                <==这里指的是分割槽的名称(label)
OS type: Linux
Block size=4096 (log=2)          <==block 的大小配置为 4K
Fragment size=4096 (log=2)
251392 inodes, 502023 blocks     <==由此配置决定的inode/block数量
25101 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=515899392
16 block groups
32768 blocks per group, 32768 fragments per group
15712 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Creating journal (8192 blocks): done <==有日志记录
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
# 这样就创建起来我们所需要的 Ext3 文件系统了！简单明了！
```

磁盘检验

fsck（file system check）用来检查和维护不一致的文件系统。

若系统掉电或磁盘发生问题，可利用fsck命令对文件系统进行检查。

语法：

```
fsck [-t 文件系统] [-ACay] 装置名称
```

选项与参数：

- -t : 给定档案系统的型式，若在 /etc/fstab 中已有定义或 kernel 本身已支援的则不需加上此参数
- -s : 依序一个一个地执行 fsck 的指令来检查
- -A : 对/etc/fstab 中所有列出来的 分区（partition）做检查
- -C : 显示完整的检查进度
- -d : 打印出 e2fsck 的 debug 结果
- -p : 同时有 -A 条件时，同时有多个 fsck 的检查一起执行
- -R : 同时有 -A 条件时，省略 / 不检查
- -V : 详细显示模式
- -a : 如果检查有错则自动修复
- -r : 如果检查有错则由使用者回答是否修复
- -y : 选项指定检测每个文件是自动输入yes，在不确定那些是不正常的时候，可以执行 # fsck -y 全部检查修复。

实例 1

查看系统有多少文件系统支持的 fsck 命令：

```
[root@www ~]# fsck[tab][tab]
fsck         fsck.cramfs  fsck.ext2    fsck.ext3    fsck.msdos   fsck.vfat
```

实例 2

强制检测 /dev/hdc6 分区:

```
[root@www ~]# fsck -C -f -t ext3 /dev/hdc6
fsck 1.39 (29-May-2006)
e2fsck 1.39 (29-May-2006)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
vbird_logical: 11/251968 files (9.1% non-contiguous), 36926/1004046 blocks
```

如果没有加上 -f 的选项，则由于这个文件系统不曾出现问题，检查的经过非常快速！若加上 -f 强制检查，才会一项一项的显示过程。

磁盘挂载与卸除

Linux 的磁盘挂载使用 `mount` 命令，卸载使用 `umount` 命令。

磁盘挂载语法：

```
mount [-t 文件系统] [-L Label名] [-o 额外选项] [-n]  装置文件名  挂载点
```

实例 1

用默认的方式，将刚刚创建的 /dev/hdc6 挂载到 /mnt/hdc6 上面！

```
[root@www ~]# mkdir /mnt/hdc6
[root@www ~]# mount /dev/hdc6 /mnt/hdc6
[root@www ~]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
.....中间省略.....
/dev/hdc6              1976312     42072   1833836   3% /mnt/hdc6
```

磁盘卸载命令 `umount` 语法：

```
umount [-fn] 装置文件名或挂载点
```

选项与参数：

- -f ：强制卸除！可用在类似网络文件系统 (NFS) 无法读取到的情况下；
- -n ：不升级 /etc/mtab 情况下卸除。

卸载/dev/hdc6

```
[root@www ~]# umount /dev/hdc6
```

Linux vi/vim

所有的 Unix Like 系统都会内建 vi 文书编辑器，其他的文书编辑器则不一定会存在。

但是目前我们使用比较多的是 vim 编辑器。

vim 具有程序编辑的能力，可以主动的以字体颜色辨别语法的正确性，方便程序设计。

什么是 vim？

Vim是从 vi 发展出来的一个文本编辑器。代码补完、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。

简单的来说， vi 是老式的字处理器，不过功能已经很齐全了，但是还是有可以进步的地方。 vim 则可以说是程序开发者的一项很好用的工具。 连 vim 的官方网站 (http://www.vim.org) 自己也说 vim 是一个程序开发工具而不是文字处理软件。

vi/vim 的使用

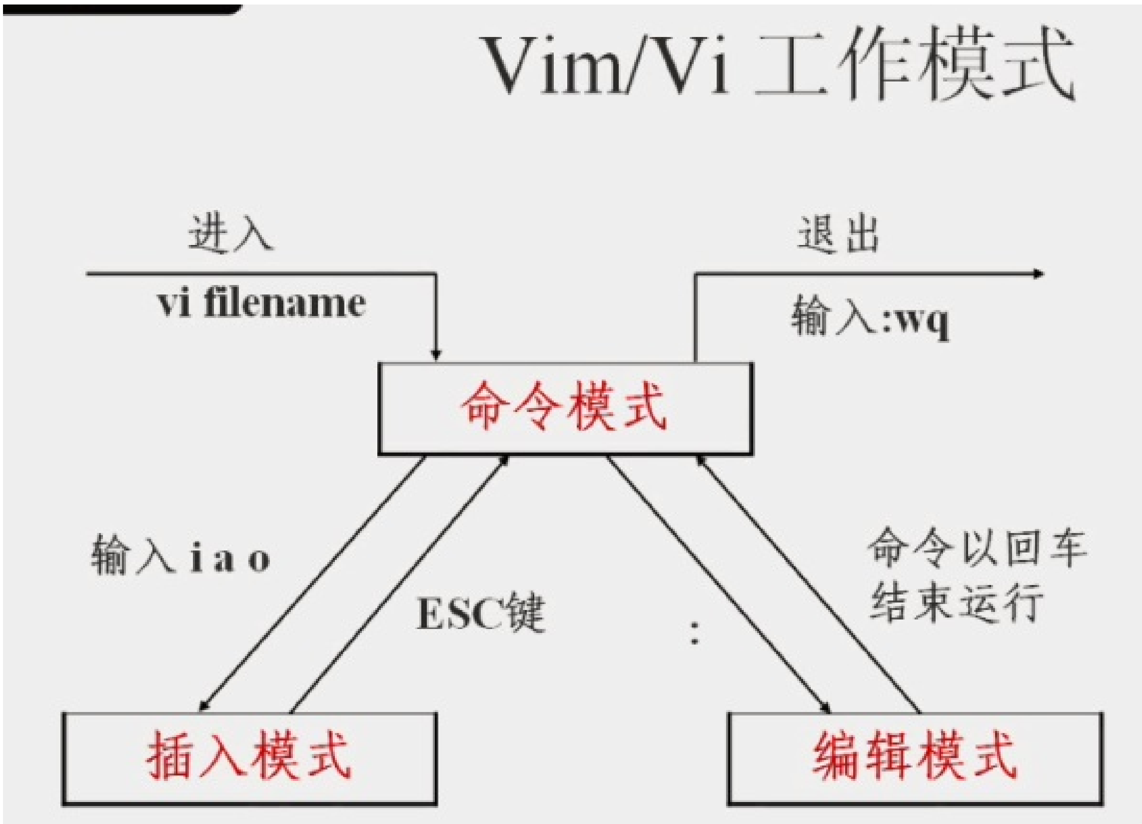
基本上 vi/vim 共分为三种模式，分别是一般模式、编辑模式与指令列命令模式。 这三种模式的作用分别是：

- 一般模式：
  以 vi 打开一个档案就直接进入一般模式了(这是默认的模式)。在这个模式中， 你可以使用『上下左右』按键来移动光标，你可以使用『删除字符』或『删除整行』来处理档案内容， 也可以使用『复制、贴上』来处理你的文件数据。

- 编辑模式：
  在一般模式中可以进行删除、复制、贴上等等的动作，但是却无法编辑文件内容的！ 要等到你按下『i, I, o, O, a, A, r, R』等任何一个字母之后才会进入编辑模式。注意了！通常在 Linux 中，按下这些按键时，在画面的左下方会出现『INSERT 或 REPLACE 』的字样，此时才可以进行编辑。而如果要回到一般模式时， 则必须要按下『Esc』这个按键即可退出编辑模式。

- 指令列命令模式：
  在一般模式当中，输入『 : / ? 』三个中的任何一个按钮，就可以将光标移动到最底下那一行。在这个模式当中， 可以提供你『搜寻资料』的动作，而读取、存盘、大量取代字符、离开 vi 、显示行号等等的动作则是在此模式中达成的！

简单的说，我们可以将这三个模式想成底下的图标来表示：



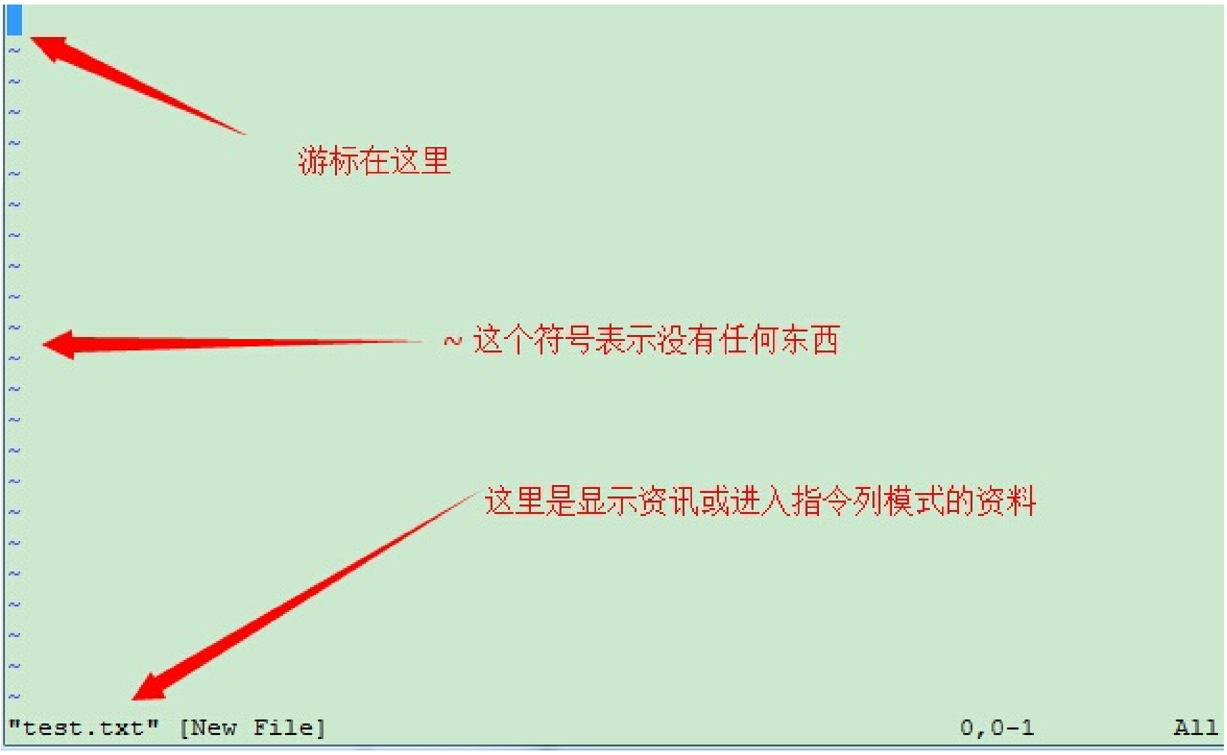
vi/vim 使用实例

使用 vi/vim 进入一般模式

如果你想要使用 vi 来建立一个名为 test.txt 的文件时，你可以这样做：

```
[root@www ~]# vi test.txt
```

直接输入 **vi 文件名** 就能够进入 vi 的一般模式了。请注意，记得 vi 后面一定要加文件名，不管该文件存在与否！

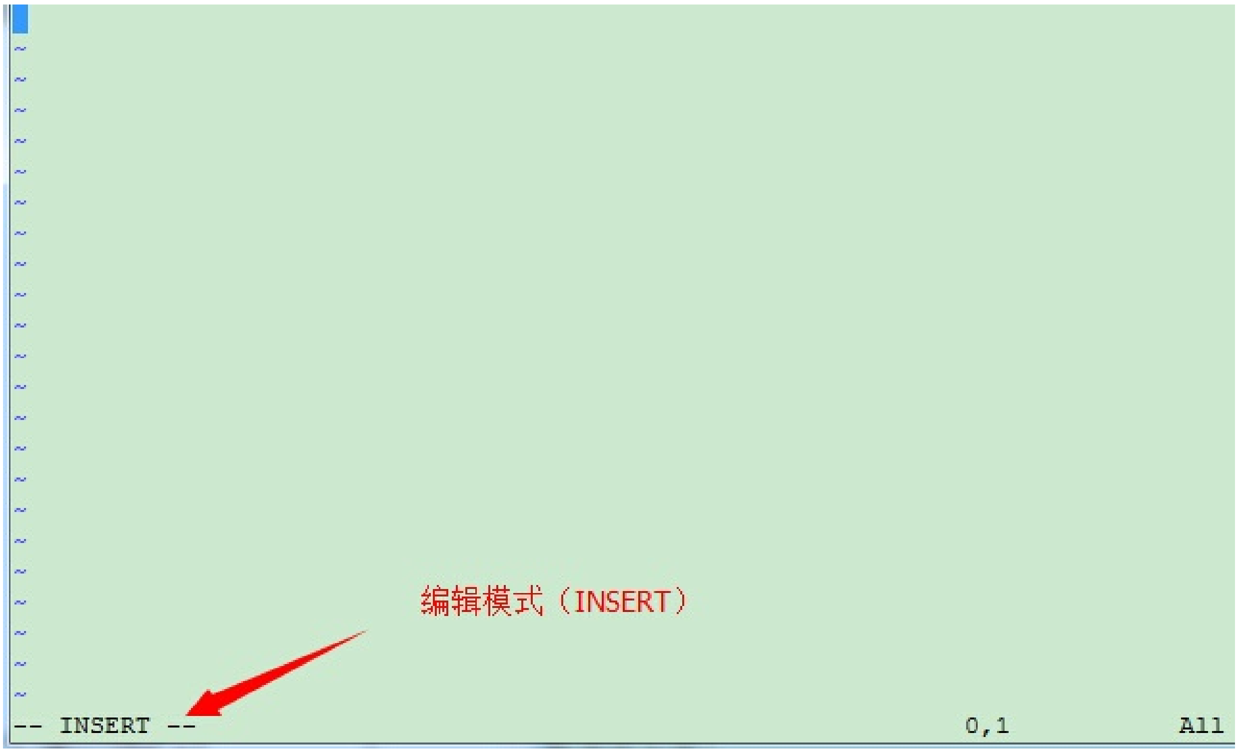


按下 i 进入编辑模式，开始编辑文字

在一般模式之中，只要按下 i, o, a 等字符就可以进入编辑模式了！

在编辑模式当中，你可以发现在左下角状态栏中会出现 –INSERT- 的字样，那就是可以输入任意字符的提示。

这个时候，键盘上除了 [Esc] 这个按键之外，其他的按键都可以视作为一般的输入按钮了，所以你可以进行任何的编辑。



按下 [ESC] 按钮回到一般模式

好了，假设我已经按照上面的样式给他编辑完毕了，那么应该要如何退出呢？是的！没错！就是给他按下 [Esc] 这个按钮即可！马上你就会发现画面左下角的 – INSERT – 不见了！

在一般模式中按下 :wq 储存后离开 vi

OK，我们要存档了，存盘并离开的指令很简单，输入『:wq』即可保存离开！

OK! 这样我们就成功创建了一个 test.txt 的文件。是不是很简单。

vi/vim 按键说明

除了上面简易范例的 i, [Esc], :wq 之外，其实 vim 还有非常多的按键可以使用。

第一部份：一般模式可用的按钮说明，光标移动、复制贴上、搜寻取代等

移动光标的方法	
h 或 向左箭头键(←)	光标向左移动一个字符
j 或 向下箭头键(↓)	光标向下移动一个字符
k 或 向上箭头键(↑)	光标向上移动一个字符
l 或 向右箭头键(→)	光标向右移动一个字符
如果你将右手放在键盘上的话，你会发现 hjkl 是排列在一起的，因此可以使用这四个按钮来移动光标。 如果想要进行	

多次移动的话，例如向下移动 30 行，可以使用 "30j" 或 "30↓" 的组合按键，亦即加上想要进行的次数(数字)后，按下动作即可！	
[Ctrl] + [f]	屏幕『向下』移动一页，相当于 [Page Down] 按键 (常用)
[Ctrl] + [b]	屏幕『向上』移动一页，相当于 [Page Up] 按键 (常用)
[Ctrl] + [d]	屏幕『向下』移动半页
[Ctrl] + [u]	屏幕『向上』移动半页
+	光标移动到非空格符的下一列
-	光标移动到非空格符的上一列
n	那个 n 表示『数字』，例如 20 。按下数字后再按空格键，光标会向右移动这一行的 n 个字符。例如 20 则光标会向后面移动 20 个字符距离。
0 或功能键[Home]	这是数字『 0 』：移动到这一行的最前面字符处 (常用)
$ 或功能键[End]	移动到这一行的最后面字符处(常用)
H	光标移动到这个屏幕的最上方那一行的第一个字符
M	光标移动到这个屏幕的中央那一行的第一个字符
L	光标移动到这个屏幕的最下方那一行的第一个字符
G	移动到这个档案的最后一行(常用)
nG	n 为数字。移动到这个档案的第 n 行。例如 20G 则会移动到这个档案的第 20 行(可配合 :set nu)
gg	移动到这个档案的第一行，相当于 1G 啊！ (常用)
n	n 为数字。光标向下移动 n 行(常用)
搜寻与取代	
/word	向光标之下寻找一个名称为 word 的字符串。例如要在档案内搜寻 vbird 这个字符串，就输入 /vbird 即可！ (常用)
?word	向光标之上寻找一个字符串名称为 word 的字符串。
	这个 n 是英文按键。代表重复前一个搜寻的

n	动作。举例来说， 如果刚刚我们执行 /vbird 去向下搜寻 vbird 这个字符串，则按下 n 后，会向下继续搜寻下一个名称为 vbird 的字符串。如果是执行 ?vbird 的话，那么按下 n 则会向上继续搜寻名称为 vbird 的字符串！
N	这个 N 是英文按键。与 n 刚好相反，为『反向』进行前一个搜寻动作。 例如 /vbird 后，按下 N 则表示『向上』搜寻 vbird 。
使用 /word 配合 n 及 N 是非常有帮助的！可以让你重复的找到一些你搜寻的关键词！	
:n1,n2s/word1/word2/g	n1 与 n2 为数字。在第 n1 与 n2 行之间寻找 word1 这个字符串，并将该字符串取代为 word2 ！举例来说，在 100 到 200 行之间搜寻 vbird 并取代为 VBIRD 则：『:100,200s/vbird/VBIRD/g』。(常用)
:1,$s/word1/word2/g	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2 ！(常用)
:1,$s/word1/word2/gc	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2 ！且在取代前显示提示字符给用户确认 (confirm) 是否需要取代！(常用)
x, X	在一行字当中，x 为向后删除一个字符 (相当于 [del] 按键)， X 为向前删除一个字符(相当于 [backspace] 亦即是退格键) (常用)
nx	n 为数字，连续向后删除 n 个字符。举例来说，我要连续删除 10 个字符， 『10x』。
dd	删除游标所在的那一整列(常用)
ndd	n 为数字。删除光标所在的向下 n 列，例如 20dd 则是删除 20 列 (常用)
d1G	删除光标所在到第一行的所有数据
dG	删除光标所在到最后一行的所有数据
d$	删除游标所在处，到该行的最后一个字符
d0	那个是数字的 0 ，删除游标所在处，到该行的最前面一个字符
yy	复制游标所在的那一行(常用)
nyy	n 为数字。复制光标所在的向下 n 列，例如 20yy 则是复制 20 列(常用)
y1G	复制游标所在列到第一列的所有数据
yG	复制游标所在列到最后一列的所有数据

y0	复制光标所在的那个字符到该行行首的所有数据
y$	复制光标所在的那个字符到该行行尾的所有数据
p, P	p 为将已复制的数据在光标下一行贴上，P 则为贴在游标上一行！ 举例来说，我目前光标在第 20 行，且已经复制了 10 行数据。则按下 p 后， 那 10 行数据会贴在原本的 20 行之后，亦即由 21 行开始贴。但如果是按下 P 呢？ 那么原本的第 20 行会被推到变成 30 行。 (常用)
J	将光标所在列与下一列的数据结合成同一列
c	重复删除多个数据，例如向下删除 10 行，[10cj]
u	复原前一个动作。(常用)
[Ctrl]+r	重做上一个动作。(常用)
这个 u 与 [Ctrl]+r 是很常用的指令！一个是复原， 另一个则是重做一次～ 利用这两个功能按键， 你的编辑，嘿嘿！很快乐的啦！	
.	不要怀疑！这就是小数点！意思是重复前一个动作的意思。 如果你想要重复删除、重复贴上等等动作，按下小数点『.』就好了！ (常用)

第二部份：一般模式切换到编辑模式的可用的按钮说明

进入插入或取代的编辑模式	
i, I	进入插入模式(Insert mode)： i 为『从目前光标所在处插入』， I 为『在目前所在行的第一个非空格符处开始插入』。 (常用)
a, A	进入插入模式(Insert mode)： a 为『从目前光标所在的下一个字符处开始插入』， A 为『从光标所在行的最后一个字符处开始插入』。(常用)
o, O	进入插入模式(Insert mode)： 这是英文字母 o 的大小写。o 为『在目前光标所在的下一行处插入新的一行』； O 为在目前光标所在处的上一行插入新的一行！(常用)
r, R	进入取代模式(Replace mode)：r 只会取代光标所在的那一个字符一次；R会一直取代光标所在的文字，直到按下 ESC 为止；(常用)
上面这些按键中，在 vi 画面的左下角处会出现『--INSERT--』或『--REPLACE--』的字样。 由名称就知道该动作了吧！！特别注意的是，我们上面也提过了，你想要在档案里面输入字符时， 一定要在左下角处看到 INSERT 或 REPLACE 才能输入喔！	
[Esc]	退出编辑模式，回到一般模式中(常用)

第三部份：一般模式切换到指令列模式的可用的按钮说明

指令列的储存、离开等指令	
:w	将编辑的数据写入硬盘档案中(常用)
:w!	若文件属性为『只读』时，强制写入该档案。不过，到底能不能写入， 还是跟你对该档案的档案权限有关啊！
:q	离开 vi (常用)
:q!	若曾修改过档案，又不想储存，使用！为强制离开不储存档案。
注意一下啊，那个惊叹号 (!) 在 vi 当中，常常具有『强制』的意思～	
:wq	储存后离开，若为 :wq! 则为强制储存后离开 (常用)
ZZ	这是大写的 Z 喔！若档案没有更动，则不储存离开，若档案已经被更动过，则储存后离开！
:w [filename]	将编辑的数据储存成另一个档案（类似另存新档）
:r [filename]	在编辑的数据中，读入另一个档案的数据。亦即将『filename』 这个档案内容加到游标所在行后面
:n1,n2 w [filename]	将 n1 到 n2 的内容储存成 filename 这个档案。
:! command	暂时离开 vi 到指令列模式下执行 command 的显示结果！例如 『:! ls /home』即可在 vi 当中察看 /home 底下以 ls 输出的档案信息！
vim 环境的变更	
:set nu	显示行号，设定之后，会在每一行的前缀显示该行的行号
:set nonu	与 set nu 相反，为取消行号！

特别注意，在 vi/vim 中，数字是很有意义的！数字通常代表重复做几次的意思！ 也有可能是代表去到第几个什么什么的意思。

举例来说，要删除 50 行，则是用 『50dd』 对吧！ 数字加在动作之前，如我要向下移动 20 行呢？那就是『20j』或者是『20↓』即可。

linux yum 命令

yum（ Yellow dog Updater, Modified）是一个在Fedora和RedHat以及SUSE中的Shell前端软件包管理器。

基於RPM包管理，能够从指定的服务器自动下载RPM包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软体包，无须繁琐地一次次下载、安装。

yum提供了查找、安装、删除某一个、一组甚至全部软件包的命令，而且命令简洁而又好记。

yum 语法

```
yum [options] [command] [package ...]
```

- **options**：可选，选项包括-h（帮助），-y（当安装过程提示选择全部为"yes"），-q（不显示安装的过程）等等。
- **command**：要进行的操作。
- **package**操作的对象。

yum常用命令

- 1.列出所有可更新的软件清单命令：yum check-update
- 2.更新所有软件命令：yum update
- 3.仅安装指定的软件命令：yum install <package_name>
- 4.仅更新指定的软件命令：yum update <package_name>
- 5.列出所有可安装的软件清单命令：yum list
- 6.删除软件包命令：yum remove <package_name>
- 7.查找软件包 命令：yum search <keyword>
- 8.清除缓存命令:
    - yum clean packages: 清除缓存目录下的软件包
    - yum clean headers: 清除缓存目录下的 headers
    - yum clean oldheaders: 清除缓存目录下旧的 headers
    - yum clean, yum clean all (= yum clean packages; yum clean oldheaders) :清除缓存目录下的软件包及旧的headers

实例 1

安装 pam-devel

```
[root@www ~]# yum install pam-devel
Setting up Install Process
Parsing package install arguments
Resolving Dependencies  <==先检查软件的属性相依问题
--> Running transaction check
---> Package pam-devel.i386 0:0.99.6.2-4.el5 set to be updated
--> Processing Dependency: pam = 0.99.6.2-4.el5 for package: pam-devel
--> Running transaction check
---> Package pam.i386 0:0.99.6.2-4.el5 set to be updated
filelists.xml.gz          100% |=========================| 1.6 MB    00:05
filelists.xml.gz          100% |=========================| 138 kB    00:00
-> Finished Dependency Resolution
……(省略)
```

实例 2

移除 pam-devel

```
[root@www ~]# yum remove pam-devel
Setting up Remove Process
Resolving Dependencies  <==同样的，先解决属性相依的问题
--> Running transaction check
---> Package pam-devel.i386 0:0.99.6.2-4.el5 set to be erased
--> Finished Dependency Resolution

Dependencies Resolved

=============================================================================
 Package                Arch       Version          Repository        Size
=============================================================================
Removing:
 pam-devel              i386       0.99.6.2-4.el5   installed         495 k

Transaction Summary
=============================================================================
Install      0 Package(s)
Update       0 Package(s)
Remove       1 Package(s)  <==还好，并没有属性相依的问题，单纯移除一个软件

Is this ok [y/N]: y
Downloading Packages:
Running rpm_check_debug
Running Transaction Test
Finished Transaction Test
Transaction Test Succeeded
Running Transaction
  Erasing   : pam-devel                    ######################### [1/1]

Removed: pam-devel.i386 0:0.99.6.2-4.el5
Complete!
```

实例 3

利用 yum 的功能，找出以 pam 为开头的软件名称有哪些？

```
[root@www ~]# yum list pam*
Installed Packages
pam.i386                  0.99.6.2-3.27.el5      installed
pam_ccreds.i386           3-5                    installed
pam_krb5.i386             2.2.14-1               installed
pam_passwdqc.i386         1.0.2-1.2.2            installed
pam_pkcs11.i386           0.5.3-23               installed
pam_smb.i386              1.1.7-7.2.1            installed
Available Packages <==底下则是『可升级』的或『未安装』的
pam.i386                  0.99.6.2-4.el5         base
pam-devel.i386            0.99.6.2-4.el5         base
pam_krb5.i386             2.2.14-10              base
```

国内 yum 源

网易（163）yum源是国内最好的yum源之一 ，无论是速度还是软件版本，都非常的不错。

将yum源设置为163 yum，可以提升软件包安装和更新的速度，同时避免一些常见软件版本无法找到。

安装步骤

首先备份/etc/yum.repos.d/CentOS-Base.repo

```
mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.backup
```

下载对应版本repo文件, 放入/etc/yum.repos.d/(操作前请做好相应备份)

- CentOS5 ：http://mirrors.163.com/.help/CentOS5-Base-163.repo
- CentOS6 ：http://mirrors.163.com/.help/CentOS6-Base-163.repo

运行以下命令生成缓存

```
yum clean all
yum makecache
```

除了网易之外，国内还有其他不错的yum源，比如中科大和搜狐。

中科大的yum源，安装方法查看：https://lug.ustc.edu.cn/wiki/mirrors/help/centos

sohu的yum源安装方法查看: http://mirrors.sohu.com/help/centos.html

Shell 编程

Shell 教程

Shell 是一个用C语言编写的程序，它是用户使用Linux的桥梁。Shell既是一种命令语言，又是一种程序设计语言。

Shell 是指一种应用程序，这个应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

Ken Thompson的sh是第一种Unix Shell，Windows Explorer是一个典型的图形界面Shell。

Shell 脚本

Shell 脚本（shell script），是一种为shell编写的脚本程序。

业界所说的shell通常都是指shell脚本，但读者朋友要知道，shell和shell script是两个不同的概念。

由于习惯的原因，简洁起见，本文出现的"shell编程"都是指shell脚本编程，不是指开发shell自身。

Shell 环境

Shell 编程跟java、php编程一样，只要有一个能编写代码的文本编辑器和一个能解释执行的脚本解释器就可以了。

Linux的Shell种类众多，常见的有：

- Bourne Shell（/usr/bin/sh或/bin/sh）
- Bourne Again Shell（/bin/bash）
- C Shell（/usr/bin/csh）
- K Shell（/usr/bin/ksh）
- Shell for Root（/sbin/sh）
- ……

本教程关注的是 Bash，也就是 Bourne Again Shell，由于易用和免费，Bash在日常工作中被广泛使用。同时，Bash也是大多数Linux系统默认的Shell。

在一般情况下，人们并不区分 Bourne Shell 和 Bourne Again Shell，所以，像 **#!/bin/sh**，它同样也可以改为**#!/bin/bash**。

`#!` 告诉系统其后路径所指定的程序即是解释此脚本文件的Shell程序。

第一个shell脚本

打开文本编辑器(可以使用vi/vim命令来创建文件)，新建一个文件test.sh，扩展名为sh（sh代表shell），扩展名并不影响脚本执行，见名知意就好，如果你用php写shell 脚本，扩展名就用php好了。

输入一些代码，第一行一般是这样：

```
#!/bin/bash
echo "Hello World !"
```

" `#!` " 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种Shell。

echo命令用于向窗口输出文本。

运行Shell脚本有两种方法：

1、作为可执行程序

将上面的代码保存为test.sh，并cd到相应目录：

```
chmod +x ./test.sh  #使脚本具有执行权限
./test.sh  #执行脚本
```

注意，一定要写成./test.sh，而不是test.sh，运行其它二进制的程序也一样，直接写test.sh，linux系统会去PATH里寻找有没有叫test.sh的，而只有/bin, /sbin, /usr/bin，/usr/sbin等在PATH里，你的当前目录通常不在PATH里，所以写成test.sh是会找不到命令的，要用./test.sh告诉系统说，就在当前目录找。

2、作为解释器参数

这种运行方式是，直接运行解释器，其参数就是shell脚本的文件名，如：

```
/bin/sh test.sh
/bin/php test.php
```

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

Shell 变量

定义变量时，变量名不加美元符号（$，PHP语言中变量需要），如：

```
your_name="w3cschool.cc"
```

注意，变量名和等号之间不能有空格，这可能和你熟悉的所有编程语言都不一样。同时，变量名的命名须遵循如下规则：

- 首个字符必须为字母（a-z，A-Z）。
- 中间不能有空格，可以使用下划线（_）。
- 不能使用标点符号。
- 不能使用bash里的关键字（可用help命令查看保留关键字）。

除了显式地直接赋值，还可以用语句给变量赋值，如：

```
for file in `ls /etc`
```

以上语句将 /etc 下目录的文件名循环出来。

使用变量

使用一个定义过的变量，只要在变量名前面加美元符号即可，如：

```
your_name="qinjx"
echo $your_name
echo ${your_name}
```

变量名外面的花括号是可选的，加不加都行，加花括号是为了帮助解释器识别变量的边界，比如下面这种情况：

```
for skill in Ada Coffe Action Java do
    echo "I am good at ${skill}Script"
done
```

如果不给skill变量加花括号，写成echo "I am good at $skillScript"，解释器就会把$skillScript当成一个变量（其值为空），代码执行结果就不是我们期望的样子了。

推荐给所有变量加上花括号，这是个好的编程习惯。

已定义的变量，可以被重新定义，如：

```
your_name="tom"
echo $your_name
your_name="alibaba"
echo $your_name
```

这样写是合法的，但注意，第二次赋值的时候不能写$your_name="alibaba"，使用变量的时候才加美元符（$）。

Shell 字符串

字符串是shell编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。单双引号的区别跟PHP类似。

单引号

```
str='this is a string'
```

单引号字符串的限制：

- 单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；
- 单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

双引号

```
your_name='qinjx'
str="Hello, I know your are \"$your_name\"! \n"
```

双引号的优点：

- 双引号里可以有变量
- 双引号里可以出现转义字符

拼接字符串

```
your_name="qinjx"
greeting="hello, "$your_name" !"
greeting_1="hello, ${your_name} !"
echo $greeting $greeting_1
```

获取字符串长度

```
string="abcd"
echo ${#string} #输出 4
```

提取子字符串

```
string="alibaba is a great company"
echo ${string:1:4} #输出liba
```

查找子字符串

```
string="alibaba is a great company"
echo `expr index "$string" is`
```

Shell 数组

bash支持一维数组（不支持多维数组），并且没有限定数组的大小。

类似与C语言，数组元素的下标由0开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于0。

定义数组

在Shell中，用括号来表示数组，数组元素用"空格"符号分割开。定义数组的一般形式为：

```
    数组名=(值1 值2 ... 值n)
```

例如：

```
array_name=(value0 value1 value2 value3)
```

或者

```
array_name=(
value0
value1
value2
value3
)
```

还可以单独定义数组的各个分量：

```
array_name[0]=value0
array_name[1]=value1
array_name[n]=valuen
```

可以不使用连续的下标，而且下标的范围没有限制。

读取数组

读取数组元素值的一般格式是：

```
    ${数组名[下标]}
```

例如：

```
valuen=${array_name[n]}
```

使用@符号可以获取数组中的所有元素，例如：

```
echo ${array_name[@]}
```

获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同，例如：

```
# 取得数组元素的个数
length=${#array_name[@]}
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]}
```

Shell 注释

以"#"开头的行就是注释，会被解释器忽略。

sh里没有多行注释，只能每一行加一个#号。只能像这样：

```
#--------------------------------------------
# 这是一个自动打ipa的脚本，基于webfrogs的ipa-build书写：
# https://github.com/webfrogs/xcode_shell/blob/master/ipa-build
# 功能：自动为etao ios app打包，产出物为14个渠道的ipa包
# 特色：全自动打包，不需要输入任何参数
#--------------------------------------------
##### 用户配置区 开始 #####
#
#
# 项目根目录，推荐将此脚本放在项目的根目录，这里就不用改了
# 应用名，确保和Xcode里Product下的target_name.app名字一致
#
##### 用户配置区 结束  #####
```

如果在开发过程中，遇到大段的代码需要临时注释起来，过一会儿又取消注释，怎么办呢？

每一行加个#符号太费力了，可以把这一段要注释的代码用一对花括号括起来，定义成一个函数，没有地方调用这个函数，这块代码就不会执行，达到了和注释一样的效果。

Shell echo命令

Shell 的 echo 指令与 PHP 的 echo 指令类似，都是用于字符串的输出。命令格式：

```
echo string
```

您可以使用echo实现更复杂的输出格式控制。

1.显示普通字符串:

```
  echo "It is a test"
```

这里的双引号完全可以省略，以下命令与上面实例效果一致：

```
  echo It is a test
```

2.显示转义字符

```
  echo "\"It is a test\""
```

结果将是:

```
  "It is a test"
```

同样，双引号也可以省略

3.显示变量

read 命令从标准输入中读取一行,并把输入行的每个字段的值指定给 shell 变量

```
#!/bin/sh
read name
echo "$name It is a test"
```

以上代码保存为 test.sh，name 接收标准输入的变量，结果将是:

```
[root@www ~]# sh test.sh
OK                     #标准输入
OK It is a test        #输出
```

4.显示换行

```
echo -e "OK!\n" # -e 开启转义
echo "It it a test"
```

输出结果：

```
OK!

It it a test
```

5.显示不换行

```
#!/bin/sh
echo -e "OK! \c" # -e 开启转义 \c 不换行
echo "It is a test"
```

输出结果：

```
OK! It is a test
```

6.显示结果定向至文件

```
echo "It is a test" > myfile
```

7.原样输出字符串，不进行转义或取变量(用单引号)

```
echo '$name\"'
```

输出结果：

```
$name\"
```

8.显示命令执行结果

```
echo `date`
```

结果将显示当前日期

```
Thu Jul 24 10:08:46 CST 2014
```

Shell test命令

Shell中的 test 命令用于检查某个条件是否成立，它可以进行数值、字符和文件三个方面的测试。

数值测试

参数	说明
-eq	等于则为真
-ne	不等于则为真
-gt	大于则为真
-ge	大于等于则为真
-lt	小于则为真
-le	小于等于则为真

实例演示：

```
num1=100
num2=100
if test $[num1] -eq $[num2]
then
    echo 'The two numbers are equal!'
else
    echo 'The two numbers are not equal!'
fi
```

输出结果：

```
The two numbers are equal!
```

字符串测试

参数	说明
=	等于则为真
!=	不相等则为真
-z 字符串	字符串长度伪则为真
-n 字符串	字符串长度不伪则为真

实例演示：

```
num1=100
num2=100
if test num1=num2
then
    echo 'The two strings are equal!'
else
    echo 'The two strings are not equal!'
fi
```

输出结果：

```
The two strings are equal!
```

文件测试

参数	说明
-e 文件名	如果文件存在则为真
-r 文件名	如果文件存在且可读则为真
-w 文件名	如果文件存在且可写则为真
-x 文件名	如果文件存在且可执行则为真
-s 文件名	如果文件存在且至少有一个字符则为真
-d 文件名	如果文件存在且为目录则为真
-f 文件名	如果文件存在且为普通文件则为真
-c 文件名	如果文件存在且为字符型特殊文件则为真
-b 文件名	如果文件存在且为块特殊文件则为真

实例演示：

```
cd /bin
if test -e ./bash
then
    echo 'The file already exists!'
else
    echo 'The file does not exists!'
fi
```

输出结果：

```
The file already exists!
```

另外，Shell还提供了与(!)、或(-o)、非(-a)三个逻辑操作符用于将测试条件连接起来，其优先级为："!"最高，"-a"次之，"-o"最低。例如：

```
cd /bin
if test -e ./notFile -o ./bash
then
    echo 'One file exists at least!'
else
    echo 'Both dose not exists!'
fi
```

输出结果：

```
One file exists at least!
```

Shell 流程控制

和Java、PHP等语言不一样，sh的流程控制不可为空，如(以下为PHP流程控制写法)：

```
<?php
if (isset($_GET["q"])) {
    search(q);
}
else {
    //do nothing
}
```

在sh/bash里可不能这么写，如果else分支没有语句执行，就不要写这个else，就像这样

if else

if

if 语句语法格式：

```
if condition
then
    command1
    command2
    ...
    commandN
fi
```

写成一行（适用于终端命令提示符）：

```
if `ps -ef | grep ssh`;  then echo hello; fi
```

末尾的fi就是if倒过来拼写，后面还会遇到类似的。

if else

if else 语法格式：

```
if condition
then
    command1
    command2
    ...
    commandN
else
    command
fi
```

if else-if else

if else-if else 语法格式：

```
if condition1
then
    command1
elif condition2
    command2
else
    commandN
fi
```

if else语句经常与test命令结合使用，如下所示：

```
num1=$[2*3]
num2=$[1+5]
if test $[num1] -eq $[num2]
then
    echo 'The two numbers are equal!'
else
    echo 'The two numbers are not equal!'
fi
```

输出结果：

```
The two numbers are equal!
```

for 循环

与其他编程语言类似，Shell支持for循环。

for循环一般格式为：

```
for var in item1 item2 ... itemN
do
    command1
    command2
    ...
    commandN
done
```

写成一行：

```
for var in item1 item2 ... itemN; do command1; command2… done;
```

当变量值在列表里，for循环即执行一次所有命令，使用变量名获取列表中的当前取值。命令可为任何有效的shell命令和语句。in列表可以包含替换、字符串和文件名。

in列表是可选的，如果不用它，for循环使用命令行的位置参数。

例如，顺序输出当前列表中的数字：

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
```

输出结果：

```
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5
```

顺序输出字符串中的字符：

```
for str in 'This is a string'
do
    echo $str
done
```

输出结果：

```
This is a string
```

while 语句

while循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。其格式为：

```
while condition
do
    command
done
```

命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。

以下是一个基本的while循环，测试条件是：如果COUNTER小于5，那么条件返回真。COUNTER从0开始，每次循环处理时，COUNTER加1。运行上述脚本，返回数字1到5，然后终止。

```
COUNTER=0
while [ $COUNTER -lt 5 ]
do
    COUNTER='expr $COUNTER+1'
    echo $COUNTER
done
```

运行脚本，输出：

```
1
2
3
4
5
```

while循环可用于读取键盘信息。下面的例子中，输入信息被设置为变量FILM，按<Ctrl-D>结束循环。

```
echo 'type <CTRL-D> to terminate'
echo -n 'enter your most liked film: ''
while read FILM
do
    echo "Yeah! great film the $FILM"
done
```

运行脚本，输出类似下面：

```
type <CTRL-D> to terminate
enter your most liked film: Sound of Music
Yeah! great film the Sound of Music
```

无限循环

无限循环语法格式：

```
while :
do
    command
done
```

或者

```
while true
do
    command
done
```

或者

```
for (( ; ; ))
```

until 循环

until循环执行一系列命令直至条件为真时停止。

until循环与while循环在处理方式上刚好相反。

一般while循环优于until循环，但在某些时候—也只是极少数情况下，until循环更加有用。

until 语法格式:

```
until condition
do
    command
done
```

条件可为任意测试条件，测试发生在循环末尾，因此循环至少执行一次—请注意这一点。

case

Shell case语句为多选择语句。可以用case语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。case语句格式如下：

```
case 值 in
模式1)
    command1
    command2
    ...
    commandN
    ;;
模式2)
    command1
    command2
    ...
    commandN
    ;;
esac
```

case工作方式如上所示。取值后面必须为单词in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。

取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。

下面的脚本提示输入1到4，与每一种模式进行匹配：

```
echo 'Input a number between 1 to 4'
echo 'Your number is:\c'
read aNum
case $aNum in
    1)  echo 'You select 1'
    ;;
    2)  echo 'You select 2'
    ;;
    3)  echo 'You select 3'
    ;;
    4)  echo 'You select 4'
    ;;
    *)  echo 'You do not select a number between 1 to 4'
    ;;
esac
```

输入不同的内容，会有不同的结果，例如：

```
Input a number between 1 to 4
Your number is:3
You select 3
```

跳出循环

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，Shell使用两个命令来实现该功能：break和continue。

break命令

break命令允许跳出所有循环（终止执行后面的所有循环）。

下面的例子中，脚本进入死循环直至用户输入数字大于5。要跳出这个循环，返回到shell提示符下，需要使用break命令。

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
        *) echo "You do not select a number between 1 to 5, game is over!"
            break
        ;;
    esac
done
```

continue

```
continue命令与break命令类似，只有一点差别，它不会跳出所有循环，仅仅跳出当前循环。
```

对上面的例子进行修改：

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
        *) echo "You do not select a number between 1 to 5!"
            continue
            echo "Game is over!"
        ;;
    esac
done
```

运行代码发现，当输入大于5的数字时，该例中的循环不会结束，语句 **echo "Game is over!"** 永远不会被执行。

esac

case的语法和C family语言差别很大，它需要一个esac（就是case反过来）作为结束标记，每个case分支用右圆括号，用两个分号表示break。

Shell 函数

linux shell 可以用户定义函数，然后在shell脚本中可以随便调用。

shell中函数的定义格式如下：

```
[ function ] funname [()]

{

    action;

    [return int;]

}
```

说明：

- 1、可以带function fun() 定义，也可以直接fun() 定义,不带任何参数。
- 2、参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。 return后跟数值n(0-255

下面的例子定义了一个函数并进行调用：

```
#!/bin/bash
demoFun(){
    echo "This is your first shell function!"
}
echo "Function begin..."
hello
echo "Function end!"
```

输出结果：

```
Function begin...
This is your first shell function!
Function end!
```

下面定义一个带有return语句的函数：

```
#!/bin/bash
funWithReturn(){
    echo "The function is to get the sum of two numbers..."
    echo -n "Input first number: "
    read aNum
    echo -n "Input another number: "
    read anotherNum
    echo "The two numbers are $aNum and $anotherNum !"
    return $(($aNum+$anotherNum))
}
funWithReturn
echo "The sum of two numbers is $? !"
```

输出类似下面：

```
The function is to get the sum of two numbers...
Input first number: 25
Input another number: 50
The two numbers are 25 and 50 !
The sum of two numbers is 75 !
```

函数返回值在调用该函数后通过 $? 来获得。

注意：所有函数在使用前必须定义。这意味着必须将函数放在脚本开始部分，直至shell解释器首次发现它时，才可以使用。调用函数仅使用其函数名即可。

函数参数

在Shell中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1表示第一个参数，$2表示第二个参数...

带参数的函数示例：

```
#!/bin/bash
funWithParam(){
    echo "The value of the first parameter is $1 !"
    echo "The value of the second parameter is $2 !"
    echo "The value of the tenth parameter is $10 !"
    echo "The value of the tenth parameter is ${10} !"
    echo "The value of the eleventh parameter is ${11} !"
    echo "The amount of the parameters is $# !"
    echo "The string of the parameters is $* !"
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

输出结果：

```
The value of the first parameter is 1 !
The value of the second parameter is 2 !
The value of the tenth parameter is 10 !
The value of the tenth parameter is 34 !
The value of the eleventh parameter is 73 !
The amount of the parameters is 12 !
The string of the parameters is 1 2 3 4 5 6 7 8 9 34 73 !"
```

注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当n>=10时，需要使用${n}来获取参数。

另外，还有几个特殊字符用来处理参数：

参数处理	说明
$#	传递到脚本的参数个数
$*	以一个单字符串显示所有向脚本传递的参数
$	脚本运行的当前进程ID号
$!	后台运行的最后一个进程的ID号
$@	与$#相同，但是使用时加引号，并在引号中返回每个参数。
$-	显示Shell使用的当前选项，与set命令功能相同。
$?	显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误。

Linux命令大全

Linux命令大全 - 文件管理

cat	chattr	chgrp	chmod
chown	cksum	cmp	diff
diffstat	file	find	git
gitview	indent	cut	ln
less	locate	lsattr	mattrib
mc	mdel	mdir	mktemp
more	mmove	mread	mren
mtools	mtoolstest	mv	od
paste	patch	rcp	rm
slocate	split	tee	tmpwatch
touch	umask	which	cp
whereis	mcopy	mshowfat	rhmask
scp	awk		

Linux cat命令

命令：cat

cat命令用于把档案串连接后传到基本输出（萤幕或加 > fileName 到另一个档案）

使用权限

所有使用者

语法格式

```
cat [-AbeEnstTuv] [--help] [--version] fileName
```

参数说明：

-n 或 --number 由 1 开始对所有输出的行数编号

-b 或 --number-nonblank 和 -n 相似，只不过对于空白行不编号

-s 或 --squeeze-blank 当遇到有连续两行以上的空白行，就代换为一行的空白行

-v 或 --show-nonprinting

实例：

把 textfile1 的档案内容加上行号后输入 textfile2 这个档案里

```
cat -n textfile1 &gt; textfile2
```

把 textfile1 和 textfile2 的档案内容加上行号（空白行不加）之后将内容附加到 textfile3 里。

```
cat -b textfile1 textfile2 &gt;&gt; textfile3
```

清空/etc/test.txt档案内容

```
cat /dev/null &gt; /etc/test.txt
```

cat 也可以用来制作镜像文件。例如要制作软碟的像文件，将软碟放好后打

```
cat /dev/fd0 &gt; OUTFILE
```

相反的，如果想把 image file 写到软碟，请打

```
cat IMG_FILE &gt; /dev/fd0
```

注：

- 1. OUTFILE 指输出的镜像文件名。
- 2. IMG_FILE 指镜像文件。
- 3. 若从镜像文件写回 device 时，device 容量需与相当。
- 4. 通常用在制作开机磁片。

Linux chattr命令

Linux chattr命令用于改变文件属性。

这项指令可改变存放在ext2文件系统上的文件或目录属性，这些属性共有以下8种模式：

1. a：让文件或目录仅供附加用途。
2. b：不更新文件或目录的最后存取时间。
3. c：将文件或目录压缩后存放。
4. d：将文件或目录排除在倾倒操作之外。
5. i：不得任意更动文件或目录。
6. s：保密性删除文件或目录。
7. S：即时更新文件或目录。
8. u：预防以外删除。

语法

```
chattr [-RV][-v<版本编号>][+/-/=<属性>][文件或目录...]
```

参数

-R 递归处理，将指定目录下的所有文件及子目录一并处理。

-v<版本编号> 设置文件或目录版本。

-V 显示指令执行过程。

+<属性> 开启文件或目录的该项属性。

-<属性> 关闭文件或目录的该项属性。

=<属性> 指定文件或目录的该项属性。

实例

用chattr命令防止系统中某个关键文件被修改：

```
chattr +i /etc/resolv.conf
```

```
lsattr /etc/resolv.conf
```

会显示如下属性

```
----i-------- /etc/resolv.conf
```

让某个文件只能往里面追加数据，但不能删除，适用于各种日志文件：

```
chattr +a /var/log/messages
```

Linux chgrp命令

Linux chgrp命令用于变更文件或目录的所属群组。

在UNIX系统家族里，文件或目录权限的掌控以拥有者及所属群组来管理。您可以使用chgrp指令去变更文件与目录的所属群组，设置方式采用群组名称或群组识别码皆可。

语法

```
chgrp [-cfhRv][--help][--version][所属群组][文件或目录...] 或 chgrp [-cfhRv][--help][--refer
```

参数说明

-c或--changes 效果类似"-v"参数，但仅回报更改的部分。

-f或--quiet或--silent 不显示错误信息。

-h或--no-dereference 只对符号连接的文件作修改，而不更动其他任何相关文件。

-R或--recursive 递归处理，将指定目录下的所有文件及子目录一并处理。

-v或--verbose 显示指令执行过程。

--help 在线帮助。

--reference=<参考文件或目录> 把指定文件或目录的所属群组全部设成和参考文件或目录的所属群组相同。

--version 显示版本信息。

实例

实例1：改变文件的群组属性：

```
chgrp -v bin log2012.log
```

输出：

```
[root@localhost test]# ll
---xrw-r-- 1 root root 302108 11-13 06:03 log2012.log
[root@localhost test]# chgrp -v bin log2012.log
```

"log2012.log" 的所属组已更改为 bin

```
[root@localhost test]# ll
---xrw-r-- 1 root bin  302108 11-13 06:03 log2012.log
```

说明： 将log2012.log文件由root群组改为bin群组

实例2：根据指定文件改变文件的群组属性

```
chgrp --reference=log2012.log log2013.log
```

输出：

```
[root@localhost test]# ll
---xrw-r-- 1 root bin  302108 11-13 06:03 log2012.log
-rw-r--r-- 1 root root     61 11-13 06:03 log2013.log
[root@localhost test]#  chgrp --reference=log2012.log log2013.log
[root@localhost test]# ll
---xrw-r-- 1 root bin  302108 11-13 06:03 log2012.log
-rw-r--r-- 1 root bin      61 11-13 06:03 log2013.log
```

说明： 改变文件log2013.log 的群组属性，使得文件log2013.log的群组属性和参考文件log2012.log的群组属性相同

Linux chmod命令

Linux/Unix 的文件调用权限分为三级 : 文件拥有者、群组、其他。利用 chmod 可以藉以控制文件如何被他人所调用。

使用权限 : 所有使用者

语法

```
chmod [-cfvR] [--help] [--version] mode file...
```

参数说明

- mode : 权限设定字串，格式如下 :

  ```
  [ugoa...][[+-=][rwxX]...][,...]
  ```

  其中 :

  - u 表示该文件的拥有者，g 表示与该文件的拥有者属于同一个群体(group)者，o 表示其他以外的人，a 表示这三者皆是。
  -
    - 表示增加权限、- 表示取消权限、= 表示唯一设定权限。
  - r 表示可读取，w 表示可写入，x 表示可执行，X 表示只有当该文件是个子目录或者该文件已经被设定过为可执行。 *-c : 若该文件权限确实已经更改，才显示其更改动作* -f : 若该文件权限无法被更改也不要显示错误讯息 *-v : 显示权限变更的详细资料* -R : 对目前目录下的所有文件与子目录进行相同的权限变更(即以递回的方式逐个变更) *--help : 显示辅助说明* --version : 显示版本

实例

将文件 file1.txt 设为所有人皆可读取 :

```
chmod ugo+r file1.txt
```

将文件 file1.txt 设为所有人皆可读取 :

```
chmod a+r file1.txt
```

将文件 file1.txt 与 file2.txt 设为该文件拥有者，与其所属同一个群体者可写入，但其他以外的人则不可写入：

```
chmod ug+w,o-w file1.txt file2.txt
```

将 ex1.py 设定为只有该文件拥有者可以执行：

```
chmod u+x ex1.py
```

将目前目录下的所有文件与子目录皆设为任何人可读取：

```
chmod -R a+r *
```

此外chmod也可以用数字来表示权限如：

```
chmod 777 file
```

语法为：

```
chmod abc file
```

其中a,b,c各为一个数字，分别表示User、Group、及Other的权限。

r=4，w=2，x=1

- 若要rwx属性则4+2+1=7；
- 若要rw-属性则4+2=6；
- 若要r-x属性则4+1=5。

```
chmod a=rwx file
```

和

```
chmod 777 file
```

效果相同

```
chmod ug=rwx,o=x file
```

和

```
chmod 771 file
```

效果相同

若用chmod 4755 filename可使此程序具有root的权限

Linux chown命令

Linux/Unix 是多人多工操作系统，所有的文件皆有拥有者。利用 chown 将指定文件的拥有者改为指定的用户或组，用户可以是用户名或者用户ID；组可以是组名或者组ID；文件是以空格分开的要改变权限的文件列表，支持通配符。 。

一般来说，这个指令只有是由系统管理者(root)所使用，一般使用者没有权限可以改变别人的文件拥有者，也没有权限可以自己的文件拥有者改设为别人。只有系统管理者(root)才有这样的权限。

使用权限 : root

语法

```
chmod [-cfhvR] [--help] [--version] user[:group] file...
```

参数 :

- user : 新的文件拥有者的使用者 ID
- group : 新的文件拥有者的使用者群体(group)
- -c : 若该文件拥有者确实已经更改，才显示其更改动作
- -f : 若该文件拥有者无法被更改也不要显示错误讯息
- -h : 只对于连结(link)进行变更，而非该 link 真正指向的文件
- -v : 显示拥有者变更的详细资料
- -R : 对目前目录下的所有文件与子目录进行相同的拥有者变更(即以递回的方式逐个变更)
- --help : 显示辅助说明
- --version : 显示版本

实例

将文件 file1.txt 的拥有者设为 users 群体的使用者 jessie :

```
chown jessie:users file1.txt
```

将目前目录下的所有文件与子目录的拥有者皆设为 users 群体的使用者 lamport :

```
chmod -R lamport:users *
```

Linux cksum命令

Linux cksum命令用于检查文件的CRC是否正确。确保文件从一个系统传输到另一个系统的过程中不被损坏。

CRC是一种排错检查方式，该校验法的标准由CCITT所指定，至少可检测到99.998%的已知错误。

指定文件交由指令"cksum"进行校验后，该指令会返回校验结果供用户核对文件是否正确无误。若不指定任何文件名称或是所给予的文件名为"-"，则指令"cksum"会从标准输入设备中读取数据。

语法

```
cksum [--help][--version][文件...]
```

参数：

- --help：在线帮助。
- --version：显示版本信息。
- 文件…:需要进行检查的文件路径

实例

使用指令"cksum"计算文件"testfile1"的完整性，输入如下命令：

```
$ cksum testfile1
```

以上命令执行后，将输出校验码等相关的信息，具体输出信息如下所示：

```
1263453430 78 testfile1         //输出信息
```

上面的输出信息中，"1263453430"表示校验码，"78"表示字节数。

注意：如果文件中有任何字符被修改，都将改变计算后CRC校验码的值。

Linux cmp命令

Linux cmp命令用于比较两个文件是否有差异。

当相互比较的两个文件完全一样时，则该指令不会显示任何信息。若发现有所差异，预设会标示出第一个不同之处的字符和列数编号。若不指定任何文件名称或是所给予的文件名为"-"，则cmp指令会从标准输入设备读取数据。

语法

```
cmp [-clsv][-i <字符数目>][--help][第一个文件][第二个文件]
```

参数：

- -c或--print-chars　除了标明差异处的十进制字码之外，一并显示该字符所对应字符。
- -i<字符数目>或--ignore-initial=<字符数目>　指定一个数目。
- -l或--verbose　标示出所有不一样的地方。
- -s或--quiet或--silent　不显示错误信息。
- -v或--version　显示版本信息。
- --help　在线帮助。

实例

要确定两个文件是否相同，请输入：

```
cmp prog.o.bak prog.o
```

这比较 prog.o.bak 和 prog.o。如果文件相同，则不显示消息。如果文件不同，则显示第一个不同的位置；例如：

```
prog.o.bak prog.o differ: char 4, line 1
```

如果显示消息 cmp: EOF on prog.o.bak，则 prog.o 的第一部分与 prog.o.bak 相同，但在 prog.o 中还有其他数据。

Linux diff命令

Linux diff命令用于比较文件的差异。

diff以逐行的方式，比较文本文件的异同处。所是指定要比较目录，则diff会比较目录中相同文件名的文件，但不会比较其中子目录。

语法

```
diff [-abBcdefHilnNpPqrstTuvwy][-<行数>][-C <行数>][-D <巨集名称>][-I <字符或字符串>][-S <文件
```

参数：

- -<行数>　指定要显示多少行的文本。此参数必须与-c或-u参数一并使用。 *-a*或*--text diff*预设只会逐行比较文本文件。 -b或--ignore-space-change　不检查空格字符的不同。
- -B或--ignore-blank-lines　不检查空白行。
- -c　显示全部内文，并标出不同之处。
- -C<行数>或--context<行数>　与执行"-c-<行数>"指令相同。
- -d或--minimal　使用不同的演算法，以较小的单位来做比较。
- -D<巨集名称>或ifdef<巨集名称>　此参数的输出格式可用于前置处理器巨集。
- -e或--ed　此参数的输出格式可用于ed的script文件。
- -f或-forward-ed　输出的格式类似ed的script文件，但按照原来文件的顺序来显示不同处。
- -H或--speed-large-files　比较大文件时，可加快速度。
- -I<字符或字符串>或--ignore-matching-lines<字符或字符串>　若两个文件在某几行有所不同，而这几行同时都包含了选项中指定的字符或字符串，则不显示这两个文件的差异。
- -i或--ignore-case　不检查大小写的不同。
- -l或--paginate　将结果交由pr程序来分页。
- -n或--rcs　将比较结果以RCS的格式来显示。
- -N或--new-file　在比较目录时，若文件A仅出现在某个目录中，预设会显示：
- Only in目录：文件A若使用-N参数，则diff会将文件A与一个空白的文件比较。
- -p　若比较的文件为C语言的程序码文件时，显示差异所在的函数名称。
- -P或--unidirectional-new-file　与-N类似，但只有当第二个目录包含了一个第一个目录所没有的文件时，才会将这个文件与空白的文件做比较。
- -q或--brief　仅显示有无差异，不显示详细的信息。
- -r或--recursive　比较子目录中的文件。

- -s或--report-identical-files　若没有发现任何差异，仍然显示信息。
- -S<文件>或--starting-file<文件>　在比较目录时，从指定的文件开始比较。
- -t或--expand-tabs　在输出时，将tab字符展开。
- -T或--initial-tab　在每行前面加上tab字符以便对齐。
- -u,-U<列数>或--unified=<列数>　以合并的方式来显示文件内容的不同。
- -v或--version　显示版本信息。
- -w或--ignore-all-space　忽略全部的空格字符。
- -W<宽度>或--width<宽度>　在使用-y参数时，指定栏宽。
- -x<文件名或目录>或--exclude<文件名或目录>　不比较选项中所指定的文件或目录。
- -X<文件>或--exclude-from<文件>　您可以将文件或目录类型存成文本文件，然后在=<文件>中指定此文本文件。
- -y或--side-by-side　以并列的方式显示文件的异同之处。
- --help　显示帮助。
- --left-column　在使用-y参数时，若两个文件某一行内容相同，则仅在左侧的栏位显示该行内容。
- --suppress-common-lines　在使用-y参数时，仅显示不同之处。

实例1：比较两个文件

```
[root@localhost test3]# diff log2014.log log2013.log
3c3
< 2014-03
---
> 2013-03
8c8
< 2013-07
---
> 2013-08
11,12d10
< 2013-11
< 2013-12
```

上面的"3c3"和"8c8"表示log2014.log和log20143log文件在3行和第8行内容有所不同；"11,12d10"表示第一个文件比第二个文件多了第11和12行。

实例2：并排格式输出

```
[root@localhost test3]# diff log2014.log log2013.log  -y -W 50
2013-01                   2013-01
2013-02                   2013-02
2014-03                 | 2013-03
2013-04                   2013-04
2013-05                   2013-05
2013-06                   2013-06
2013-07                   2013-07
2013-07                 | 2013-08
2013-09                   2013-09
2013-10                   2013-10
2013-11                 <
2013-12                 <
[root@localhost test3]# diff log2013.log log2014.log  -y -W 50
2013-01                   2013-01
2013-02                   2013-02
2013-03                 | 2014-03
2013-04                   2013-04
2013-05                   2013-05
2013-06                   2013-06
2013-07                   2013-07
2013-08                 | 2013-07
2013-09                   2013-09
2013-10                   2013-10
                        > 2013-11
                        > 2013-12
```

说明：

- "|"表示前后2个文件内容有不同
- "<"表示后面文件比前面文件少了1行内容
- ">"表示后面文件比前面文件多了1行内容

Linux diffstat命令

Linux diffstat命令根据diff的比较结果，显示统计数字。

diffstat读取diff的输出结果，然后统计各文件的插入，删除，修改等差异计量。

语法

```
diff [-wV][-n <文件名长度>][-p <文件名长度>]
```

参数：

- -n<文件名长度>　指定文件名长度，指定的长度必须大于或等于所有文件中最长的文件名。
- -p<文件名长度>　与-n参数相同，但此处的<文件名长度>包括了文件的路径。
- -w　指定输出时栏位的宽度。
- -V　显示版本信息。

实例

用户也可以直接使用"|"将diff指令所输出的结果直接送给diffstat指令进行统计结果的显示。

使用该指令时，若所比较的文件或者子目录不在当前目录下，则应该使用其完整路径。

将目录"test1"和"test2"下的同名文件"testf.txt"使用diff指令进行比较。然后使用diffstat指令对结果进行统计显示，输入如下命令：

```
$ diff test1 test2 | diffstat   #进行比较结果的统计显示
```

注意：使用这条命令可以非常方便地实现统计显示的功能。

对于查看文件中的内容，用户可以通过指令"cat"进行查看即可，具体操作如下：

```
$ cat test1/testf.txt           #查看test1/testf的内容
abc
def
ghi
jkl
mno
pqr
stu
vws
$ cat test2/testf.txt           #查看test2/testf的内容
abc
def
ghi
jkl
mno
```

从上面的文件内容显示，可以看到两个文件内容的差别。现在来运行刚才的命令，对文件比较的结果进行统计显示，结果如下：

```
testfile | 2 +-             #统计信息输出显示
1 file changed, 1 insertion(+), 1 deletion(-)
```

Linux file命令

Linux file命令用于辨识文件类型。

通过file指令，我们得以辨识该文件的类型。

语法

```
file [-beLvz][-f <名称文件>][-m <魔法数字文件>...][文件或目录...]
```

参数：

- -b 　列出辨识结果时，不显示文件名称。
- -c 　详细显示指令执行过程，便于排错或分析程序执行的情形。
- -f<名称文件> 　指定名称文件，其内容有一个或多个文件名称呢感，让file依序辨识这些文件，格式为每列一个文件名称。
- -L 　直接显示符号连接所指向的文件的类别。
- -m<魔法数字文件> 　指定魔法数字文件。
- -v 　显示版本信息。
- -z 　尝试去解读压缩文件的内容。
- [文件或目录...] 要确定类型的文件列表，多个文件之间使用空格分开，可以使用shell通配符匹配多个文件。

实例

显示文件类型：

```
[root@localhost ~]# file install.log
install.log: UTF-8 Unicode text

[root@localhost ~]# file -b install.log      <== 不显示文件名称
UTF-8 Unicode text

[root@localhost ~]# file -i install.log      <== 显示MIME类别。
install.log: text/plain; charset=utf-8

[root@localhost ~]# file -b -i install.log
text/plain; charset=utf-8
```

显示符号链接的文件类型

```
[root@localhost ~]# ls -l /var/mail
lrwxrwxrwx 1 root root 10 08-13 00:11 /var/mail -> spool/mail

[root@localhost ~]# file /var/mail
/var/mail: symbolic link to `spool/mail'

[root@localhost ~]# file -L /var/mail
/var/mail: directory

[root@localhost ~]# file /var/spool/mail
/var/spool/mail: directory

[root@localhost ~]# file -L /var/spool/mail
/var/spool/mail: directory
```

Linux find命令

Linux find命令用来在指定目录下查找文件。任何位于参数之前的字符串都将被视为欲查找的目录名。如果使用该命令时，不设置任何参数，则find命令将在当前目录下查找子目录与文件。并且将查找到的子目录和文件全部进行显示。

语法

```
find   path   -option   [   -print ]   [ -exec   -ok   command ]   {} ;
```

参数说明 :

find 根据下列规则判断 path 和 expression，在命令列上第一个 - ( ) , ! 之前的部份为 path，之后的是 expression。如果 path 是空字串则使用目前路径，如果 expression 是空字串则使用 -print 为预设 expression。

expression 中可使用的选项有二三十个之多，在此只介绍最常用的部份。

-mount, -xdev : 只检查和指定目录在同一个文件系统下的文件，避免列出其它文件系统中的文件

-amin n : 在过去 n 分钟内被读取过

-anewer file : 比文件 file 更晚被读取过的文件

-atime n : 在过去 n 天过读取过的文件

-cmin n : 在过去 n 分钟内被修改过

-cnewer file :比文件 file 更新的文件

-ctime n : 在过去 n 天过修改过的文件

-empty : 空的文件-gid n or -group name : gid 是 n 或是 group 名称是 name

-ipath p, -path p : 路径名称符合 p 的文件，ipath 会忽略大小写

-name name, -iname name : 文件名称符合 name 的文件。iname 会忽略大小写

-size n : 文件大小 是 n 单位，b 代表 512 位元组的区块，c 表示字元数，k 表示 kilo bytes，w 是二个位元组。-type c : 文件类型是 c 的文件。

d: 目录

c: 字型装置文件

b: 区块装置文件

p: 具名贮列

f: 一般文件

l: 符号连结

s: socket

-pid n : process id 是 n 的文件

你可以使用 ( ) 将运算式分隔，并使用下列运算。

exp1 -and exp2

! expr

-not expr

exp1 -or exp2

exp1, exp2

实例

将目前目录及其子目录下所有延伸档名是 c 的文件列出来。

```
# find . -name "*.c"
```

将目前目录其其下子目录中所有一般文件列出

```
# find . -ftype f
```

将目前目录及其子目录下所有最近 20 分钟内更新过的文件列出

```
# find . -ctime -20
```

查找/var/logs目录中更改时间在7日以前的普通文件，并在删除之前询问它们：

```
$ find /var/logs -type f -mtime +7 -ok rm { } ;
```

查找前目录中文件属主具有读、写权限，并且文件所属组的用户和其他用户具有读权限的文件：

```
$ find . -type f -perm 644 -exec ls -l { } ;
```

为了查找系统中所有文件长度为0的普通文件，并列出它们的完整路径：

```
$ find / -type f -size 0 -exec ls -l { } ;
```

查找/var/logs目录中更改时间在7日以前的普通文件，并在删除之前询问它们:

```
$ find /var/logs -type f -mtime +7 -ok rm { } ;
```

Linux git命令

Linux git命令是文字模式下的文件管理员。

git是用来管理文件的程序，它十分类似DOS下的Norton Commander，具有互动式操作界面。它的操作方法和Norton Commander几乎一样。

语法

```
git
```

操作说明：

- F1 ：执行info指令，查询指令相关信息，会要求您输入欲查询的名称。
- F2 ：执行cat指令，列出文件内容。
- F3 ：执行gitview指令，观看文件内容。
- F4 ：执行vi指令，编辑文件内容。
- F5 ：执行cp指令，复制文件或目录，会要求您输入目标文件或目录。
- F6 ：执行mv指令，移动文件或目录，或是更改其名称，会要求您输入目标文件或目录。
- F7 ：执行mkdir指令，建立目录。
- F8 ：执行rm指令，删除文件或目录。
- F9 ：执行make指令，批处理执行指令或编译程序时，会要求您输入相关命令。
- F10 ：离开git文件管理员。

Linux gitview命令

Linux gitview命令用于观看文件的内容，它会同时显示十六进制和ASCII格式的字码。

语法

```
gitview [-bchilv][文件]
```

参数：

- -b　单色模式，不使用ANSI控制码显示彩色。
- -c　彩色模式，使用ANSI控制码显示色彩。
- -h　在线帮助。
- -i　显示存放gitview程序的所在位置。
- -l　不使用先前的显示字符。
- -v　显示版本信息。

实例

使用指令gitview以彩色模式观看文件"/home/ rootlocal/demo.txt"中的内容，输入如下命令：

```
$ gitview -c /home/rootlocal/demo.txt      #使用gitview指令观看指定文件内容
```

Linux indent命令

Linux indent命令用于调整C原始代码文件的格式。

indent可辨识C的原始代码文件，并加以格式化，以方便程序设计师阅读。

语法

```
indent [参数][源文件] 或 indent [参数][源文件][-o 目标文件]
```

参数：

- -bad或--blank-lines-after-declarations 　在声明区段或加上空白行。
- -bap或--blank-lines-after-procedures 　在程序或加上空白行。
- -bbb或--blank-lines-after-block-comments 　在注释区段后加上空白行。
- -bc或--blank-lines-after-commas 　在声明区段中，若出现逗号即换行。
- -bl或--braces-after-if-line 　if(或是else,for等等)与后面执行区段的"{"不同行，且"}"自成一行。
- -bli<缩排格数>或--brace-indent<缩排格数> 　设置{ }缩排的格数。
- -br或--braces-on-if-line 　if(或是else,for等等)与后面执行跛段的"{"不同行，且"}"自成一行。
- -bs或--blank-before-sizeof 　在sizeof之后空一格。
- -c<栏数>或--comment-indentation<栏数> 　将注释置于程序码右侧指定的栏位。
- -cd<栏数>或--declaration-comment-column<栏数> 　将注释置于声明右侧指定的栏位。
- -cdb或--comment-delimiters-on-blank-lines 　注释符号自成一行。
- -ce或--cuddle-else 　将else置于"}"(if执行区段的结尾)之后。
- -ci<缩排格数>或--continuation-indentation<缩排格数> 　叙述过长而换行时，指定换行后缩排的格数。
- -cli<缩排格数>或--case-indentation-<缩排格数> 　使用case时，switch缩排的格数。
- -cp<栏数>或-else-endif-column<栏数> 　将注释置于else与elseif叙述右侧定的栏位。
- -cs或--space-after-cast 　在cast之后空一格。
- -d<缩排格数>或-line-comments-indentation<缩排格数> 　针对不是放在程序码右侧的注释，设置其缩排格数。
- -di<栏数>或--declaration-indentation<栏数> 　将声明区段的变量置于指定的栏位。
- -fc1或--format-first-column-comments 　针对放在每行最前端的注释，设置其格式。
- -fca或--format-all-comments 　设置所有注释的格式。
- -gnu或--gnu-style 　指定使用GNU的格式，此为预设值。
- -i<格数>或--indent-level<格数> 　设置缩排的格数。
- -ip<格数>或--parameter-indentation<格数> 　设置参数的缩排格数。

- -kr或--k-and-r-style　指定使用Kernighan&Ritchie的格式。
- -lp或--continue-at-parentheses　叙述过长而换行，且叙述中包含了括弧时，将括弧中的每行起始栏位内容垂直对其排列。
- -nbad或--no-blank-lines-after-declarations　在声明区段后不要加上空白行。
- -nbap或--no-blank-lines-after-procedures　在程序后不要加上空白行。
- -nbbb或--no-blank-lines-after-block-comments　在注释区段后不要加上空白行。
- -nbc或--no-blank-lines-after-commas　在声明区段中，即使出现逗号，仍旧不要换行。
- -ncdb或--no-comment-delimiters-on-blank-lines　注释符号不要自成一行。
- -nce或--dont-cuddle-else　不要将else置于"}"之后。
- -ncs或--no-space-after-casts　不要在cast之后空一格。
- -nfc1或--dont-format-first-column-comments　不要格式化放在每行最前端的注释。
- -nfca或--dont-format-comments　不要格式化任何的注释。
- -nip或--no-parameter-indentation　参数不要缩排。
- -nlp或--dont-line-up-parentheses　叙述过长而换行，且叙述中包含了括弧时，不用将括弧中的每行起始栏位垂直对其排列。
- -npcs或--no-space-after-function-call-names　在调用的函数名称之后，不要加上空格。
- -npro或--ignore-profile　不要读取indent的配置文件.indent.pro。
- -npsl或--dont-break-procedure-type　程序类型与程序名称放在同一行。
- -nsc或--dont-star-comments　注解左侧不要加上星号(*)。
- -nsob或--leave-optional-semicolon　不用处理多余的空白行。
- -nss或--dont-space-special-semicolon　若for或while区段仅有一行时，在分号前不加上空格。
- -nv或--no-verbosity　不显示详细的信息。
- -orig或--original　使用Berkeley的格式。
- -pcs或--space-after-procedure-calls　在调用的函数名称与"{"之间加上空格。
- -psl或--procnames-start-lines　程序类型置于程序名称的前一行。
- -sc或--start-left-side-of-comments　在每行注释左侧加上星号(*)。
- -sob或--swallow-optional-blank-lines　删除多余的空白行。
- -ss或--space-special-semicolon　若for或swile区段今有一行时，在分号前加上空格。
- -st或--standard-output　将结果显示在标准输出设备。
- -T　数据类型名称缩排。
- -ts<格数>或--tab-size<格数>　设置tab的长度。
- -v或--verbose　执行时显示详细的信息。
- -version　显示版本信息。

Indent代码格式化说明

使用的**indent**参数	值	含义
--blank-lines-after-declarations	bad	变量声明后加空行

--blank-lines-after-procedures	bap	函数结束后加空行
--blank-lines-before-block-comments	bbb	块注释前加空行
--break-before-boolean-operator	bbo	较长的行，在逻辑运算符前分行
--blank-lines-after-commas	nbc	变量声明中，逗号分隔的变量不分行
--braces-after-if-line	bl	"if"和"{"分做两行
--brace-indent 0	bli0	"{"不继续缩进
--braces-after-struct-decl-line	bls	定义结构，"struct"和"{"分行
--comment-indentationn	c33	语句后注释开始于行33
--declaration-comment-columnn	cd33	变量声明后注释开始于行33
--comment-delimiters-on-blank-lines	ncdb	不将单行注释变为块注释
--cuddle-do-while	ncdw	"do --- while"的"while"和其前面的"}"另起一行
--cuddle-else	nce	"else"和其前面的"}"另起一行
--case-indentation 0	cli0	switch中的case语句所进0个空格
--else-endif-columnn	cp33	#else, #endif后面的注释开始于行33
--space-after-cast	cs	在类型转换后面加空格
--line-comments-indentation n	d0	单行注释（不从1列开始的），不向左缩进
--break-function-decl-args	nbfda	关闭：函数的参数一个一行
--declaration-indentationn	di2	变量声明，变量开始于2行，即不必对齐
--format-first-column-comments	nfc1	不格式化起于第一行的注释
--format-all-comments	nfca	不开启全部格式化注释的开关
--honour-newlines	hnl	Prefer to break long lines at the position of newlines in the input.
--indent-leveln	i4	设置缩进多少字符，如果为tab的整数倍，用tab来缩进，否则用空格填充。
--parameter-indentationn	ip5	旧风格的函数定义中参数说明缩进5个空格
--line-length 75	l75	非注释行最长75

--continue-at-parentheses	lp	续行从上一行出现的括号开始
--space-after-procedure-calls	pcs	函数和"("之间插入一个空格
--space-after-parentheses	nprs	在"（"后"）"前不插入空格
--procnames-start-lines	psl	将函数名和返回类型放在两行定义
--space-after-for	saf	for后面有空格
--space-after-if	sai	if后面有空格
--space-after-while	saw	while后面有空格
--start-left-side-of-comments	nsc	不在生成的块注释中加*
--swallow-optional-blank-lines	nsob	不去掉可添加的空行
--space-special-semicolon	nss	一行的for或while语句，在";"前不加空。
--tab-size	ts4	一个tab为4个空格（要能整除"-in"）
--use-tabs	ut	使用tab来缩进

Linux cut命令

Linux cut命令用于显示每行从开头算起 num1 到 num2 的文字。

语法

```
cut  [-bn] [file]
cut [-c] [file]
cut [-df] [file]
```

使用说明:

cut 命令从文件的每一行剪切字节、字符和字段并将这些字节、字符和字段写至标准输出。

如果不指定 File 参数，cut 命令将读取标准输入。必须指定 -b、-c 或 -f 标志之一。

参数:

- -b ：以字节为单位进行分割。这些字节位置将忽略多字节字符边界，除非也指定了 -n 标志。
- -c ：以字符为单位进行分割。
- -d ：自定义分隔符，默认为制表符。
- -f ：与-d一起使用，指定显示哪个区域。
- -n ：取消分割多字节字符。仅和 -b 标志一起使用。如果字符的最后一个字节落在由 -b 标志的 List 参数指示的
  范围之内，该字符将被写出；否则，该字符将被排除

实例

当你执行who命令时，会输出类似如下的内容：

```
$ who
rocrocket :0           2009-01-08 11:07
rocrocket pts/0        2009-01-08 11:23 (:0.0)
rocrocket pts/1        2009-01-08 14:15 (:0.0)
```

如果我们想提取每一行的第3个字节，就这样：

```
$ who|cut -b 3
c
c
```

Linux ln命令

Linux ln命令是一个非常重要命令，它的功能是为某一个文件在另外一个位置建立一个同步的链接。

当我们需要在不同的目录，用到相同的文件时，我们不需要在每一个需要的目录下都放一个必须相同的文件，我们只要在某个固定的目录，放上该文件，然后在 其它的目录下用ln命令链接（link）它就可以，不必重复的占用磁盘空间。

语法

```
ln [参数][源文件或目录][目标文件或目录]
```

其中参数的格式为

[-bdfinsvF] [-S backup-suffix] [-V {numbered,existing,simple}]

[--help] [--version] [--]

命令功能 :
Linux文件系统中，有所谓的链接(link)，我们可以将其视为档案的别名，而链接又可分为两种 : 硬链接(hard link)与软链接(symbolic link)，硬链接的意思是一个档案可以有多个名称，而软链接的方式则是产生一个特殊的档案，该档案的内容是指向另一个档案的位置。硬链接是存在同一个文件系统中，而软链接却可以跨越不同的文件系统。

不论是硬链接或软链接都不会将原本的档案复制一份，只会占用非常少量的磁碟空间。

软链接：

- 1.软链接，以路径的形式存在。类似于Windows操作系统中的快捷方式
- 2.软链接可以 跨文件系统 ，硬链接不可以
- 3.软链接可以对一个不存在的文件名进行链接
- 4.软链接可以对目录进行链接

硬链接：

- 1.硬链接，以文件副本的形式存在。但不占用实际空间。
- 2.不允许给目录创建硬链接
- 3.硬链接只有在同一个文件系统中才能创建

命令参数

必要参数：

- -b 删除，覆盖以前建立的链接
- -d 允许超级用户制作目录的硬链接
- -f 强制执行
- -i 交互模式，文件存在则提示用户是否覆盖
- -n 把符号链接视为一般目录
- -s 软链接(符号链接)
- -v 显示详细的处理过程

选择参数：

- -S "-S<字尾备份字符串> "或 "--suffix=<字尾备份字符串>"
- -V "-V<备份方式>"或"--version-control=<备份方式>"
- --help 显示帮助信息
- --version 显示版本信息

实例

给文件创建软链接，为log2013.log文件创建软链接link2013，如果log2013.log丢失，link2013将失效：

```
ln -s log2013.log link2013
```

输出：

```
[root@localhost test]# ll
-rw-r--r-- 1 root bin      61 11-13 06:03 log2013.log
[root@localhost test]# ln -s log2013.log link2013
[root@localhost test]# ll
lrwxrwxrwx 1 root root     11 12-07 16:01 link2013 -> log2013.log
-rw-r--r-- 1 root bin      61 11-13 06:03 log2013.log
```

给文件创建硬链接，为log2013.log创建硬链接ln2013，log2013.log与ln2013的各项属性相同

```
ln log2013.log ln2013
```

输出：

```
[root@localhost test]# ll
lrwxrwxrwx 1 root root     11 12-07 16:01 link2013 -> log2013.log
-rw-r--r-- 1 root bin      61 11-13 06:03 log2013.log
[root@localhost test]# ln log2013.log ln2013
[root@localhost test]# ll
lrwxrwxrwx 1 root root     11 12-07 16:01 link2013 -> log2013.log
-rw-r--r-- 2 root bin      61 11-13 06:03 ln2013
-rw-r--r-- 2 root bin      61 11-13 06:03 log2013.log
```

Linux less命令

less 与 more 类似，但使用 less 可以随意浏览文件，而 more 仅能向前移动，却不能向后移动，而且 less 在查看之前不会加载整个文件。

语法

```
less [参数] 文件
```

参数说明：

- -b <缓冲区大小> 设置缓冲区的大小
- -e 当文件显示结束后，自动离开
- -f 强迫打开特殊文件，例如外围设备代号、目录和二进制文件
- -g 只标志最后搜索的关键词
- -i 忽略搜索时的大小写
- -m 显示类似more命令的百分比
- -N 显示每行的行号
- -o <文件名> 将less 输出的内容在指定文件中保存起来
- -Q 不使用警告音
- -s 显示连续空行为一行
- -S 行过长时间将超出部分舍弃
- -x <数字> 将"tab"键显示为规定的数字空格
- /字符串：向下搜索"字符串"的功能
- ?字符串：向上搜索"字符串"的功能
- n：重复前一个搜索（与 / 或 ? 有关）
- N：反向重复前一个搜索（与 / 或 ? 有关）
- b 向后翻一页
- d 向后翻半页
- h 显示帮助界面
- Q 退出less 命令
- u 向前滚动半页
- y 向前滚动一行
- 空格键 滚动一行
- 回车键 滚动一页
- [pagedown]： 向下翻动一页
- [pageup]： 向上翻动一页

实例

1、查看文件

```
less log2013.log
```

2、ps查看进程信息并通过less分页显示

```
ps -ef |less
```

3、查看命令历史使用记录并通过less分页显示

```
[root@localhost test]# history | less
22  scp -r tomcat6.0.32 root@192.168.120.203:/opt/soft
23  cd ..
24  scp -r web root@192.168.120.203:/opt/
25  cd soft
26  ls
……省略……
```

4、浏览多个文件

```
less log2013.log log2014.log
```

说明：
输入 ：n后，切换到 log2014.log
输入 ：p 后，切换到log2013.log

附加备注

1.全屏导航

- ctrl + F - 向前移动一屏
- ctrl + B - 向后移动一屏
- ctrl + D - 向前移动半屏
- ctrl + U - 向后移动半屏

2.单行导航

- j - 向前移动一行
- k - 向后移动一行

3.其它导航

- G - 移动到最后一行
- g - 移动到第一行

- q / ZZ - 退出 less 命令

4.其它有用的命令

- v - 使用配置的编辑器编辑当前文件
- h - 显示 less 的帮助文档
- &pattern - 仅显示匹配模式的行，而不是整个文件

5.标记导航

当使用 less 查看大文件时，可以在任何一个位置作标记，可以通过命令导航到标有特定标记的文本位置：

- ma - 使用 a 标记文本的当前位置
- 'a - 导航到标记 a 处

Linux locate命令

Linux locate命令用于查找符合条件的文档，他会去保存文档和目录名称的数据库内，查找合乎范本样式条件的文档或目录。

一般情况我们只需要输入 **locate your_file_name** 即可查找指定文件。

语法

```
locate [-d ][--help][--version][范本样式...]
```

参数：

- -d或--database= 配置locate指令使用的数据库。locate指令预设的数据库位于/var/lib/slocate目录里，文档名为slocate.db，您可使用 这个参数另行指定。
- --help 　在线帮助。
- --version 　显示版本信息。

实例

查找passwd文件，输入以下命令：

```
locate passwd
```

附加说明

locate与find 不同: find 是去硬盘找，locate 只在/var/lib/slocate资料库中找。

locate的速度比find快，它并不是真的查找，而是查数据库，一般文件数据库在/var/lib/slocate/slocate.db中，所以locate的查找并不是实时的，而是以数据库的更新为准，一般是系统自己维护，也可以手工升级数据库 ，命令为：

```
locate -u
```

Linux lsattr命令

Linux lsattr命令用于显示文件属性。

用chattr执行改变文件或目录的属性，可执行lsattr指令查询其属性。

语法

```
lsattr [-adlRvV][文件或目录...]
```

参数：

- -a 显示所有文件和目录，包括以"."为名称开头字符的额外内建，现行目录"."与上层目录".."。
- -d 显示，目录名称，而非其内容。
- -l 此参数目前没有任何作用。
- -R 递归处理，将指定目录下的所有文件及子目录一并处理。
- -v 显示文件或目录版本。
- -V 显示版本信息。

实例

1、用chattr命令防止系统中某个关键文件被修改：

```
# chattr +i /etc/resolv.conf
```

然后用mv /etc/resolv.conf等命令操作于该文件，都是得到Operation not permitted 的结果。

vim编辑该文件时会提示W10: Warning: Changing a readonly file错误。要想修改此文件就要把i属性去掉：

```
chattr -i /etc/resolv.conf
```

使用 lsattr 命令来显示文件属性：

```
# lsattr /etc/resolv.conf
```

输出结果为:

```
----i-------- /etc/resolv.conf
```

2、让某个文件只能往里面追加数据，但不能删除，适用于各种日志文件：

```
# chattr +a /var/log/messages
```

Linux mattrib命令

Linux mattrib命令用来变更或显示MS-DOS文件的属性。

mattrib为mtools工具指令，模拟MS-DOS的attrib指令，可变更MS-DOS文件的属性。

语法

```
mattrib [-a|+a] [-h|+h] [-r|+r] [-s|+s] [-/] [-X] msdosfile [ msdosfiles ... ]
```

参数：

- -a/+a 除去/设定备份属性。
- -h/+h 除去/设定隐藏属性。
- -r/+r 除去/设定唯读属性。
- -s/+s 除去/设定系统属性。
- -/ 递回的处理包含所有子目录下的档案。
- -X 以较短的格式输出结果。

实例

列出 A 槽 MSDOS 格式磁片上所有文件的属性。

```
mattrib a:
```

除去 A 槽磁片上 msdos.sys 档案的隐藏、系统与唯读属性。

```
mattrib -h -s -r a:msdos.sys
```

除去 A 槽磁片上包含子目录下所有档案的唯读属性。

```
mattrib -r -/ a:*.*
```

Linux mc命令

Linux mc命令用于提供一个菜单式的文件管理程序。

执行mc之后，将会看到菜单式的文件管理程序，共分成４个部分。

语法

```
mc [-abcdfhkPstuUVx][-C <参数>][-l <文件>][-v <文件>][目录]
```

参　　数：

- -a 　当mc程序画线时不用绘图字符画线。
- -b 　使用单色模式显示。
- -c 　使用彩色模式显示。
- -C<参数> 　指定显示的颜色。
- -d 　不使用鼠标。
- -f 　显示mc函数库所在的目录。
- -h 　显示帮助。
- -k 　重设softkeys成预设置。
- -l<文件> 　在指定文件中保存ftpfs对话窗的内容。
- -P 　程序结束时，列出最后的工作目录。
- -s 　用慢速的终端机模式显示，在这模式下将减少大量的绘图及文字显示。
- -t 　使用TEMPCAP变量设置终端机，而不使用预设置。
- -u 　不用目前的shell程序。
- -U 　使用目前的shell程序。
- -v<文件> 　使用mc的内部编辑器来显示指定的文件。
- -V 　显示版本信息。
- -x 　指定以xterm模式显示。

Linux MC 相关操作

命令按键	描 述
F9 or Esc+9	激活菜单栏
Tab	在两个窗口间移动
F10 or Esc+0	退出MC
Control-Enter or Alt-Enter	可以将文件名拷贝到命令行
F1 or Esc+1	打开帮助页面

虽然MC很好用，不过我还是建议大家使用命令行工具！

Linux mdel命令

Linux mdel命令用来删除 MSDOS 格式的档案。

在删除只读之前会有提示信息产生。

语法

```
mdel [-v] msdosfile [ msdosfiles ... ]
```

参数：

- -v 显示更多的讯息。

实例

将 A 槽磁片根目录中的 autoexec.bat 删除。

```
mdel a:autoexec.bat .
```

Linux mdir命令

Linux mdir命令用于显示MS-DOS目录。

mdir为mtools工具指令，模拟MS-DOS的dir指令，可显示MS-DOS文件系统中的目录内容。

语法

```
mdir [-afwx/][目录]
```

参数：

- -/ 显示目录下所有子目录与文件。
- -a 　显示隐藏文件。
- -f 　不显示磁盘所剩余的可用空间。
- -w 　仅显示目录或文件名称，并以横排方式呈现，以便一次能显示较多的目录或文件。
- -X 　仅显示目录下所有子目录与文件的完整路径，不显示其他信息。

实例

显示a盘中的内容

```
$ mdir -/ a:\*
```

以上命令执行后，mdir将显示指定盘"a:\"中的所有子目录及其中的文件信息，如下所示：

```
Volume in drive A has no label  #加载信息
Volume Serial Number is 13D2~055C
Directory for A:\                    #以下为目录信息
./TEST <DIR> 2011-08-23 16:59
#显示格式为文件名，目录大小，修改时间
AUTORUN.INF 265 2011-08-23 16:53
AUTORUN.BAT 43 2011-08-23 16:56
3 files 308 bytes              #统计总大小
724 325 bytes free             #剩余空间
```

Linux mktemp命令

Linux mktemp命令用于建立暂存文件。

mktemp建立的一个暂存文件，供shell script使用。

语法

```
mktemp [-qu][文件名参数]
```

参数：

- -q 执行时若发生错误，不会显示任何信息。
- -u 暂存文件会在mktemp结束前先行删除。
- [文件名参数] 文件名参数必须是以"自订名称.XXXXXX"的格式。

实例

使用mktemp 命令生成临时文件时，文件名参数应当以"文件名.XXXX"的形式给出，mktemp 会根据文件名参数建立一个临时文件。在命令行提示符输入如下命令：

```
mktemp tmp.xxxx #生成临时文件
```

使用该命令后，可使用dir 或ls看当前目录，得到如下结果：

```
cmd@cmd-desktop:~$ mktemp tmp.xxxx #生成临时文件
cmd@cmd-desktop:~$dir #查看当前目录
file test testfile testfile1 tmp.3847 #生成了tmp.3847
```

由此可见，生成的临时文件为tmp.3847，其中，文件名参数中的"XXXX"被4 个随机产生的字符所取代。

Linux more命令

Linux more 命令类似 cat ，不过会以一页一页的形式显示，更方便使用者逐页阅读，而最基本的指令就是按空白键（space）就往下一页显示，按 b 键就会往回（back）一页显示，而且还有搜寻字串的功能（与 vi 相似），使用中的说明文件，请按 h 。

语法

```
more [-dlfpcsu] [-num] [+/pattern] [+linenum] [fileNames..]
```

参数：

- -num 一次显示的行数
- -d 提示使用者，在画面下方显示 [Press space to continue, 'q' to quit.] ，如果使用者按错键，则会显示 [Press 'h' for instructions.] 而不是 '哔' 声
- -l 取消遇见特殊字元 ^L（送纸字元）时会暂停的功能
- -f 计算行数时，以实际上的行数，而非自动换行过后的行数（有些单行字数太长的会被扩展为两行或两行以上）
- -p 不以卷动的方式显示每一页，而是先清除萤幕后再显示内容
- -c 跟 -p 相似，不同的是先显示内容再清除其他旧资料
- -s 当遇到有连续两行以上的空白行，就代换为一行的空白行
- -u 不显示下引号 （根据环境变数 TERM 指定的 terminal 而有所不同）
- +/pattern 在每个文档显示前搜寻该字串（pattern），然后从该字串之后开始显示
- +num 从第 num 行开始显示
- fileNames 欲显示内容的文档，可为复数个数

实例

逐页显示 testfile 文档内容，如有连续两行以上空白行则以一行空白行显示。

```
more -s testfile
```

从第 20 行开始显示 testfile 之文档内容。

```
more +20 testfile
```

常用操作命令

- Enter 向下n行，需要定义。默认为1行

- Ctrl+F 向下滚动一屏
- 空格键 向下滚动一屏
- Ctrl+B 返回上一屏
- = 输出当前行的行号
- ：f 输出文件名和当前行的行号
- V 调用vi编辑器
- !命令 调用Shell，并执行命令
- q 退出more

Linux mmove命令

Linux mmove命令用于在MS-DOS文件系统中，移动文件或目录，或更改名称。

mmove为mtools工具命令，模拟MS-DOS的move命令，可在MS-DOS文件系统中移动现有的文件或目录，或是更改现有文件或目录的名称。

语法

```
mmove [源文件或目录...][目标文件或目录]
```

参数说明:

- [源文件或目录...]: 执行操作的源文件或目录路径
- [目标文件或目录]: 执行操作后的目标文件或目录路径

实例

使用指令mmove将文件"autorun.bat"移动到目录"test"中，输入如下命令：

```
$ mmove autorun.bat test        #移动文件到目录test中
```

以上命令执行以后，指令mmove会将文件"autorun.bat"移动到指定目录"test"中。

注意：用户可以使用mdir指令查看移动后的文件或目录信息。

Linux mread命令

Linux mread命令用于将MS-DOS文件复制到Linux/Unix的目录中。

mread为mtools工具命令，可将MS-DOS文件复制到Linux的文件系统中。这个命令目前已经不常用，一般都使用mcopy命令来代替。

语法

```
mread [MS-DOS文件...][Linux文件或目录]
```

参数说明:

- [MS-DOS文件...]: 执行操作的DOS源文件或目录路径
- [Linux文件或目录]: 执行操作后的Linux目标文件或目录路径

实例

使用指令mread将盘"a:\"中的所有内容复制到当前工作目录下，输入如下命令：

```
$ mread a:\* ./     #将a盘上的所有文件复制到当前工作目录
```

执行该命令前，可以先使用mdir命令查看原来的目录结构。执行mread之后，可使用ls命令再次查看复制之后的文件结构，结果如下所示：

```
$ mdir -/ a:\*      #查看a盘中的文件
Volume in drive A has no label          #加载信息
Volume Serial Number is 13D2~055C
Directory for A:/                   #以下为目录信息
./TEST <DIR> 2011-08-23 16:59
#显示格式为文件名，目录大小，修改时间
AUTORUN.INF 265 2011-08-23 16:53
AUTORUN.BAT 43 2011-08-23 16:56
3 files 308 bytes                       #统计总大小
724 325 bytes free                  #剩余空间
$ mread A:\* ./     #将a盘上所有文件复制到当前工作目录
$ ls                                    #查看文件或子目录信息
TEST AUTORUN.INF AUTORUN.BAT            #显示复制后的内容
```

Linux mren命令

Linux mren命令用于更改MS-DOS文件或目录的名称，或是移动文件或目录。

mren为MS-DOS工具指令，与DOS下的ren指令相似，可以实现更改MS-DOS文件或目录名称。

源文件必须是磁盘上已经存在的文件，若忽略盘符及路径，则表示当前盘及当前目录的文件。

新文件名是所要更换的文件名称。新文件名称前不可以加与源文件不同的盘符及路径，因为该命令只能更改同一盘上的文件名称。

语法

```
mren [源文件或目录...][目标文件或目录]
```

参数说明：

- [源文件或目录...]：执行操作的源文件名或者源文件路径

实例

使用指令mren将a盘下的文件"autorun.bat"的文件名修改为"auto.bat"，输入如下命令：

```
$ mren a:\autorun.bat auto.bat
#将文件autorun.bat重命名为auto.bat
```

使用该命令前后使用mdir命令查看并对比，得到结果如下：

```
$ mdir -/ a:\*                    #查看a盘中的文件
Volume in drive A has no label  #加载信息
Volume Serial Number is 13D2~055C
Directory for A:\                   #以下为目录信息
./TEST <DIR> 2011-08-23 16:59       #文件名，目录大小，修改时间
AUTORUN.BAT 43 2011-08-23 16:56
3 files 308 bytes                   #统计总大小
724 325 bytes free              #剩余空间
#将文件autorun.bat重命名为auto.bat
$ mren a:\autorun.bat auto.bat
$ mdir -/ a:\*                    #再次查看a盘中文件
Volume in drive A has no label      #加载信息
Volume Serial Number is 13D2~055C
Directory for A:\                   #以下为目录信息
./TEST <DIR> 2011-08-23 16:59       #文件名目录大小 修改时间
#文件名被改为auto.bat，修改时间改为当前系统时间
AUTO.BAT 43 2011-08-23 16:56
3 files 308 bytes                   #统计总大小
724 325 bytes free              #剩余空间
```

Linux mtools命令

Linux mtools命令用于显示mtools支持的指令。

mtools为MS-DOS文件系统的工具程序，可模拟许多MS-DOS的指令。这些指令都是mtools的符号连接，因此会有一些共同的特性。

语法

```
mtools
```

参数说明：

- -a 　长文件名重复时自动更改目标文件的长文件名。
- -A 　短文件名重复但长文件名不同时自动更改目标文件的短文件名。
- -o 　长文件名重复时，将目标文件覆盖现有的文件。
- -O 　短文件名重复但长文件名不同时，将目标文件覆盖现有的文件。
- -r 　长文件名重复时，要求用户更改目标文件的长文件名。
- -R 　短文件名重复但长文件名不同时，要求用户更改目标文件的短文件名。
- -s 　长文件名重复时，则不处理该目标文件。
- -S 　短文件名重复但长文件名不同时，则不处理该目标文件。
- -v 　执行时显示详细的说明。
- -V 　显示版本信息。

实例

显示 mtools软件包所支持的MS-DOS命令。

在命令提示符中直接输入mtools，可显示其所支持的MS-DOS命令，如下所示：

```
$ mtools #显示所支持的MS-DOS命令
Supported commands: #命令列表
mattrib, mbadblocks, mcat, mcd, mclasserase, mcopy, mdel, mdeltree
mdir, mdoctorfat, mdu, mformat, minfo, mlabel, mmd, mmount
mpartition, mrd, mread, mmove, mren, mshowfat, mtoolstest, mtype
mwrite, mzip
```

Linux mtoolstest命令

Linux mtoolstest命令用于测试并显示mtools的相关设置。

mtoolstest为mtools工具指令，可读取与分析mtools的配置文件，并在屏幕上显示结果。

语法

```
mtoolstest
```

实例

在命令行中直接输入mtoolstest，即可显示mtools软件包当前的配置信息，结果如下：

```
$ mtoolstest #显示mtools 软件包当前的配置信息
drive J: #mtools软件包当前的配置信息列表
#fn=0 mode=0 builtin
file="/dev/sdb4" fat_bits=16
tracks=0 heads=0 sectors=0 hidden=0
offset=0x0
partition=0
mformat_only
drive Z:
#fn=0 mode=0 builtin
file="/dev/sdb4" fat_bits=16
tracks=0 heads=0 sectors=0 hidden=0
offset=0x0
partition=0
mformat_only
drive X:
#fn=0 mode=0 builtin
file="$DISPLAY" fat_bits=0
tracks=0 heads=0 sectors=0 hidden=0
offset=0x0
partition=0
drive A:
#fn=2 mode=128 defined in /etc/mtools.conf
file="/dev/fd0" fat_bits=0
tracks=0 heads=0 sectors=0 hidden=0
offset=0x0
partition=0
exclusive
drive B:
#fn=2 mode=128 defined in /etc/mtools.conf
file="/dev/fd1" fat_bits=0
tracks=0 heads=0 sectors=0 hidden=0
offset=0x0
partition=0
exclusive
drive M:
#fn=2 mode=0 defined in /etc/mtools.conf
file="/var/lib/dosemu/hdimage.first" fat_bits=0
tracks=0 heads=0 sectors=0 hidden=0
offset=0x80
partition=1
drive N:
#fn=2 mode=0 defined in /etc/mtools.conf
file="/var/lib/dosemu/fdimage" fat_bits=0
tracks=0 heads=0 sectors=0 hidden=0
offset=0x0
partition=0
mtools_fat_compatibility=0
mtools_skip_check=0
mtools_lower_case=0
```

Linux mv命令

Linux mv命令用来为文件或目录改名、或将文件或目录移入其它位置。

语法

```
mv [options] source dest
mv [options] source... directory
```

参数说明：

- -i: 若指定目录已有同名文件，则先询问是否覆盖旧文件;
- -f: 在mv操作要覆盖某已有的目标文件时不给任何指示;

mv参数设置与运行结果

命令格式	运行结果
mv 文件名 文件名	将源文件名改为目标文件名
mv 文件名 目录名	将文件移动到目标目录
mv 目录名 目录名	目标目录已存在，将源目录 移动到目标目录；目标 目录不存在则改名
mv 目录名 文件名	出错

实例

将文件 aaa 更名为 bbb :

```
mv aaa bbb
```

将info目录放入logs目录中。注意，如果logs目录不存在，则该命令将info改名为logs。

```
mv info/ logs
```

再如将/usr/student下的所有文件和目录移到当前目录下，命令行为：

```
$ mv /usr/student/*  .
```

Linux od命令

Linux od命令用于输出文件内容。

od指令会读取所给予的文件的内容，并将其内容以八进制字码呈现出来。

语法

```
od [-abcdfhilovx][-A <字码基数>][-j <字符数目>][-N <字符数目>][-s <字符串字符数>][-t <输出格式>]
```

参数：

- -a 此参数的效果和同时指定"-ta"参数相同。
- -A<字码基数> 选择要以何种基数计算字码。
- -b 此参数的效果和同时指定"-toC"参数相同。
- -c 此参数的效果和同时指定"-tC"参数相同。
- -d 此参数的效果和同时指定"-tu2"参数相同。
- -f 此参数的效果和同时指定"-tfF"参数相同。
- -h 此参数的效果和同时指定"-tx2"参数相同。
- -i 此参数的效果和同时指定"-td2"参数相同。
- -j<字符数目>或--skip-bytes=<字符数目> 略过设置的字符数目。
- -l 此参数的效果和同时指定"-td4"参数相同。
- -N<字符数目>或--read-bytes=<字符数目> 到设置的字符数目为止。
- -o 此参数的效果和同时指定"-to2"参数相同。
- -s<字符串字符数>或--strings=<字符串字符数> 只显示符合指定的字符数目的字符串。
- -t<输出格式>或--format=<输出格式> 设置输出格式。
- -v或--output-duplicates 输出时不省略重复的数据。
- -w<每列字符数>或--width=<每列字符数> 设置每列的最大字符数。
- -x 此参数的效果和同时指定"-h"参数相同。
- --help 在线帮助。
- --version 显示版本信息。

实例

创建 tmp 文件：

```
$ echo abcdef g > tmp
$ cat tmp
abcdef g
```

使用 od 命令：

```
$ od -b tmp
0000000 141 142 143 144 145 146 040 147 012
0000011
```

使用单字节八进制解释进行输出，注意左侧的默认地址格式为八字节:

```
$ od -c tmp
0000000   a   b   c   d   e   f       g  \n
0000011
```

使用ASCII码进行输出，注意其中包括转义字符

```
$ od -t d1 tmp
0000000   97   98   99  100  101  102   32  103   10
0000011
```

使用单字节十进制进行解释

```
$ od -A d -c tmp
0000000   a   b   c   d   e   f       g  \n
0000009
```

Linux paste命令

Linux paste命令用于合并文件的列。

paste指令会把每个文件以列对列的方式，一列列地加以合并。

语法

```
paste [-s][-d <间隔字符>][--help][--version][文件...]
```

参数：

- -d<间隔字符>或--delimiters=<间隔字符>　用指定的间隔字符取代跳格字符。
- -s或--serial　串列进行而非平行处理。
- --help　在线帮助。
- --version　显示帮助信息。
- [文件…] 指定操作的文件路径

实例

使用paste指令将文件"file"、"testfile"、"testfile1"进行合并，输入如下命令：

```
paste file testfile testfile1 #合并指定文件的内容
```

但是，在执行以上命令之前，首先使用"cat"指令对3个文件内容进行查看，显示如下所示：

```
$ cat file                  #file文件的内容
xiongdan 200
lihaihui 233
lymlrl 231
$ cat testfile              #testfile文件的内容
liangyuanm  ss
$ cat testfile1             #testfile1文件的内容
huanggai 56
zhixi 73
```

当合并指令"$ paste file testfile testfile1"执行后，程序界面中将显示合并后的文件内容，如下所示：

```
xiongdan 200
lihaihui 233
lymlrl 231
liangyuanm  ss
huanggai 56
zhixi 73
```

若使用paste指令的参数"-s"，则可以将一个文件中的多行数据合并为一行进行显示。例如，将文件"file"中的3行数据合并为一行数据进行显示，输入如下命令

```
$ paste -s file              #合并指定文件的多行数据
```

上面的命令执行后，显示的数据内容如下所示：

```
xiongdan 200 lihaihui 233 lymlrl 231
```

注意：参数"-s"只是将testfile文件的内容调整显示方式，并不会改变原文件的内容格式。

Linux patch命令

Linux patch命令用于修补文件。

patch指令让用户利用设置修补文件的方式，修改，更新原始文件。倘若一次仅修改一个文件，可直接在指令列中下达指令依序执行。如果配合修补文件的方式则能一次修补大批文件，这也是Linux系统核心的升级方法之一。

语法

```
patch [-bceEflnNRstTuvZ][-B <备份字首字符串>][-d <工作目录>][-D <标示符号>][-F <监别列数>][-g <
```

参数：

- -b或--backup　备份每一个原始文件。
- -B<备份字首字符串>或--prefix=<备份字首字符串>　设置文件备份时，附加在文件名称前面的字首字符串，该字符串可以是路径名称。
- -c或--context　把修补数据解译成关联性的差异。
- -d<工作目录>或--directory=<工作目录>　设置工作目录。
- -D<标示符号>或--ifdef=<标示符号>　用指定的符号把改变的地方标示出来。
- -e或--ed　把修补数据解译成ed指令可用的叙述文件。
- -E或--remove-empty-files　若修补过后输出的文件其内容是一片空白，则移除该文件。
- -f或--force　此参数的效果和指定"-t"参数类似，但会假设修补数据的版本为新　版本。
- -F<监别列数>或--fuzz<监别列数>　设置监别列数的最大值。
- -g<控制数值>或--get=<控制数值>　设置以RSC或SCCS控制修补作业。
- -i<修补文件>或--input=<修补文件>　读取指定的修补问家你。
- -l或--ignore-whitespace　忽略修补数据与输入数据的跳格，空格字符。
- -n或--normal　把修补数据解译成一般性的差异。
- -N或--forward　忽略修补的数据较原始文件的版本更旧，或该版本的修补数据已使　用过。
- -o<输出文件>或--output=<输出文件>　设置输出文件的名称，修补过的文件会以该名称存放。
- -p<剥离层级>或--strip=<剥离层级>　设置欲剥离几层路径名称。
- -f<拒绝文件>或--reject-file=<拒绝文件>　设置保存拒绝修补相关信息的文件名称，预设的文件名称为.rej。
- -R或--reverse　假设修补数据是由新旧文件交换位置而产生。
- -s或--quiet或--silent　不显示指令执行过程，除非发生错误。
- -t或--batch　自动略过错误，不询问任何问题。

- -T或--set-time　此参数的效果和指定"-Z"参数类似，但以本地时间为主。
- -u或--unified　把修补数据解译成一致化的差异。
- -v或--version　显示版本信息。
- -V<备份方式>或--version-control=<备份方式>　用"-b"参数备份目标文件后，备份文件的字尾会被加上一个备份字符串，这个字符串不仅可用"-z"参数变更，当使用"-V"参数指定不同备份方式时，也会产生不同字尾的备份字符串。
- -Y<备份字首字符串>或--basename-prefix=--<备份字首字符串>　设置文件备份时，附加在文件基本名称开头的字首字符串。
- -z<备份字尾字符串>或--suffix=<备份字尾字符串>　此参数的效果和指定"-B"参数类似，差别在于修补作业使用的路径与文件名若为src/linux/fs/super.c，加上"backup/"字符串后，文件super.c会备份于/src/linux/fs/backup目录里。
- -Z或--set-utc　把修补过的文件更改，存取时间设为UTC。
- --backup-if-mismatch　在修补数据不完全吻合，且没有刻意指定要备份文件时，才备份文件。
- --binary　以二进制模式读写数据，而不通过标准输出设备。
- --help　在线帮助。
- --nobackup-if-mismatch　在修补数据不完全吻合，且没有刻意指定要备份文件时，不要备份文件。
- --verbose　详细显示指令的执行过程。

实例

使用patch指令将文件"testfile1"升级，其升级补丁文件为"testfile.patch"，输入如下命令：

```
$ patch -p0 testfile1 testfile.patch    #使用补丁程序升级文件
```

使用该命令前，可以先使用指令"cat"查看"testfile1"的内容。在需要修改升级的文件与原文件之间使用指令"diff"比较可以生成补丁文件。具体操作如下所示：

```
$ cat testfile1                 #查看testfile1的内容
Hello,This is the firstfile!
$ cat testfile2                 #查看testfile2的内容
Hello,Thisisthesecondfile!
$ diff testfile1 testfile2          #比较两个文件
1c1
<Hello,Thisisthefirstfile!
---
>Hello,Thisisthesecondfile!
#将比较结果保存到tetsfile.patch文件
$ diff testfile1 testfile2>testfile.patch
$ cat testfile.patch                #查看补丁包的内容
1c1
<Hello,Thisisthefirstfile!
---
>Hello,Thisisthesecondfile!
#使用补丁包升级testfile1文件
$ patch -p0 testfile1 testfile.patch
patching file testfile1
$cat testfile1                  #再次查看testfile1的内容
#testfile1文件被修改为与testfile2一样的内容
Hello,This is the secondfile!
```

注意：上述命令代码中，"$ diff testfile1 testfile2>testfile. patch"所使用的操作符"＞"表示将该操作符左边的文件数据写入到右边所指向的文件中。在这里，即是指将两个文件比较后的结果写入到文件"testfile.patch"中。

Linux rcp命令

Linux rcp命令用于复制远程文件或目录。

rcp指令用在远端复制文件或目录，如同时指定两个以上的文件或目录，且最后的目的地是一个已经存在的目录，则它灰把前面指定的所有文件或目录复制到该目录中。

语法

```
rcp [-pr][源文件或目录][目标文件或目录]
```

或

```
rcp [-pr][源文件或目录...][目标文件]
```

参数：

-p 　保留源文件或目录的属性，包括拥有者，所属群组，权限与时间。

-r 　递归处理，将指定目录下的文件与子目录一并处理。

实例

使用rcp指令复制远程文件到本地进行保存。

设本地主机当前账户为rootlocal，远程主机账户为root，要将远程主机（218.6.132.5）主目录下的文件"testfile"复制到本地目录"test"中，则输入如下命令：

```
rcp root@218.6.132.5:./testfile testfile  #复制远程文件到本地
rcp root@218.6.132.5:home/rootlocal/testfile testfile
#要求当前登录账户cmd 登录到远程主机
rcp 218.6.132.5:./testfile testfile
```

注意：指令"rcp"执行以后不会有返回信息，仅需要在目录"test"下查看是否存在文件"testfile"。若存在，则表示远程复制操作成功，否则远程复制操作失败。

Linux rm命令

Linux rm命令用于删除一个文件或者目录。

语法

```
rm [options] name...
```

参数：

- -i 删除前逐一询问确认。
- -f 即使原档案属性设为唯读，亦直接删除，无需逐一确认。
- -r 将目录及以下之档案亦逐一删除。

实例

删除文件可以直接使用rm命令，若删除目录则必须配合选项"-r"，例如：

```
# rm  test.txt
rm：是否删除 一般文件 "test.txt"? y
# rm  homework
rm: 无法删除目录"homework": 是一个目录
# rm  -r  homework
rm：是否删除 目录 "homework"? y
```

删除当前目录下的所有文件及目录，命令行为：

```
rm  -r  *
```

文件一旦通过rm命令删除，则无法恢复，所以必须格外小心地使用该命令。

Linux slocate命令

Linux slocate命令查找文件或目录。

slocate本身具有一个数据库，里面存放了系统中文件与目录的相关信息。

语法

```
slocate [-u][--help][--version][-d <目录>][查找的文件]
```

参数：

- -d<目录>或--database=<目录> 　指定数据库所在的目录。
- -u 　更新slocate数据库。
- --help 　显示帮助。
- --version 　显示版本信息。

实例

使用指令"slocate"显示文件名中含有关键字"fdisk"的文件路径信息，输入如下命令：

```
$ slocate fdisk #显示文件名中含有fdisk关键字的文件的路径信息
```

执行以上命令后，指令执行的输出信息如下：

```
$ slocate fdisk #显示文件名中含有fdisk 关键字的文件的路径信息
/root/cfdisk        #搜索到的文件路径列表
/root/fdisk
/root/sfdisk
/usr/include/grub/ieee1275/ofdisk.h
/usr/share/doc/util-Linux/README.cfdisk
/usr/share/doc/util-Linux/README.fdisk.gz
/usr/share/doc/util-Linux/examples/sfdisk.examples.gz
```

Linux split命令

Linux split命令用于将一个文件分割成数个。

该指令将大文件分割成较小的文件，在默认情况下将按照每1000行切割成一个小文件。

语法

```
split [--help][--version][-<行数>][-b <字节>][-C <字节>][-l <行数>][要切割的文件][输出文件名]
```

参数说明：

- -<行数> : 指定每多少行切成一个小文件
- -b<字节> : 指定每多少字节切成一个小文件
- --help : 在线帮助
- --version : 显示版本信息
- -C<字节> : 与参数"-b"相似，但是在切 割时将尽量维持每行的完整性
- [输出文件名] : 设置切割后文件的前置文件名， split会自动在前置文件名后再加上编号

实例

使用指令"split"将文件"README"每6行切割成一个文件，输入如下命令：

```
$ split -6 README       #将README文件每六行分割成一个文件
```

以上命令执行后，指令"split"会将原来的大文件"README"切割成多个以"x"开头的小文件。而在这些小文件中，每个文件都只有6行内容。

使用指令"ls"查看当前目录结构，如下所示：

```
$ ls                                #执行ls指令
#获得当前目录结构
README xaa xad xag xab xae xah xac xaf xai
```

Linux tee命令

Linux tee命令用于读取标准输入的数据，并将其内容输出成文件。

tee指令会从标准输入设备读取数据，将其内容输出到标准输出设备，同时保存成文件。

语法

```
tee [-ai][--help][--version][文件...]
```

参数：

- -a或--append 附加到既有文件的后面，而非覆盖它．
- -i或--ignore-interrupts 忽略中断信号。
- --help 在线帮助。
- --version 显示版本信息。

实例

使用指令"tee"将用户输入的数据同时保存到文件"file1"和"file2"中，输入如下命令：

```
$ tee file1 file2                   #在两个文件中复制内容
```

以上命令执行后，将提示用户输入需要保存到文件的数据，如下所示：

```
My Linux                        #提示用户输入数据
My Linux                        #输出数据，进行输出反馈
```

此时，可以分别打开文件"file1"和"file2"，查看其内容是否均是"My Linux"即可判断指令"tee"是否执行成功。

Linux tmpwatch命令

Linux tmpwatch命令用于删除暂存文件。

执行tmpwatch指令可删除不必要的暂存文件，您可以设置文件超期时间，单位以小时计算。

语法

```
tmpwatch [-afqv][--test][超期时间][目录...]
```

参数：

- -a或--all 删除任何类型的文件。
- -f或--force 强制删除文件或目录，其效果类似rm指令的"-f"参数。
- -q或--quiet 不显示指令执行过程。
- -v或--verbose 详细显示指令执行过程。
- -test 仅作测试，并不真的删除文件或目录。

实例

使用指令"tmpwatch"删除目录"/tmp"中超过一天未使用的文件，输入如下命令：

```
$ tmpwatch 24 /tmp/ #删除/tmp目录中超过一天未使用的文件
```

以上命令执行后，其执行结果如下所示：

```
removing directctmp/orbit-tom if not empty
```

注意：该指令需要root权限，因此在使用tmpwatch命令前应该使用su命令切换用户。切换管理权限操作如下所示：

```
$ su                              #切换到root用户
口令：**********                  #输入用户密码
```

Linux touch命令

Linux touch命令用于修改文件或者目录的时间属性，包括存取时间和更改时间。若文件不存在，系统会建立一个新的文件。

ls -l 可以显示档案的时间记录。

语法

```
touch [-acfm][-d<日期时间>][-r<参考文件或目录>] [-t<日期时间>][--help][--version][文件或目录…]
```

- 参数说明：
- a 改变档案的读取时间记录。
- m 改变档案的修改时间记录。
- c 假如目的档案不存在，不会建立新的档案。与 --no-create 的效果一样。
- f 不使用，是为了与其他 unix 系统的相容性而保留。
- r 使用参考档的时间记录，与 --file 的效果一样。
- d 设定时间与日期，可以使用各种不同的格式。
- t 设定档案的时间记录，格式与 date 指令相同。
- --no-create 不会建立新档案。
- --help 列出指令格式。
- --version 列出版本讯息。

实例

使用指令"touch"修改文件"testfile"的时间属性为当前系统时间，输入如下命令：

```
$ touch testfile                #修改文件的时间属性
```

首先，使用ls命令查看testfile文件的属性，如下所示：

```
$ ls -l testfile                #查看文件的时间属性
#原来文件的修改时间为16:09
-rw-r--r-- 1 hdd hdd 55 2011-08-22 16:09 testfile
```

执行指令"touch"修改文件属性以后，并再次查看该文件的时间属性，如下所示：

```
$ touch testfile                #修改文件时间属性为当前系统时间
$ ls -l testfile                #查看文件的时间属性
#修改后文件的时间属性为当前系统时间
-rw-r--r-- 1 hdd hdd 55 2011-08-22 19:53 testfile
```

使用指令"touch"时，如果指定的文件不存在，则将创建一个新的空白文件。例如，在当前目录下，使用该指令创建一个空白文件"file"，输入如下命令：

```
$ touch file            #创建一个名为“file”的新的空白文件
```

Linux umask命令

Linux umask命令指定在建立文件时预设的权限掩码。

umask可用来设定[权限掩码]。[权限掩码]是由3个八进制的数字所组成，将现有的存取权限减掉权限掩码后，即可产生建立文件时预设的权限。

语法

```
umask [-S][权限掩码]
```

参数说明：

-S　以文字的方式来表示权限掩码。

实例

使用指令"umask"查看当前权限掩码，则输入下面的命令：

```
$ umask                         #获取当前权限掩码
```

执行上面的指令后，输出信息如下：

```
0022
```

接下来，使用指令"mkdir"创建一个目录，并使用指令"ls"获取该目录的详细信息，输入命令如下：

```
$ mkdir test1                        #创建目录
$ ls –d –l test1/                    #显示目录的详细信息
```

执行上面的命令后，将显示新创建目录的详细信息，如下所示：

```
drwxr-xr-x 2 rootlocal rootlocal 4096 2011-9-19 21:46 test1/
```

注意：在上面的输出信息中，"drwxr-xr-x"="777-022=755"。

Linux which命令

Linux which命令用于查找文件。

which指令会在环境变量$PATH设置的目录里查找符合条件的文件。

语法

```
which [文件...]
```

参数：

- -n<文件名长度>　指定文件名长度，指定的长度必须大于或等于所有文件中最长的文件名。
- -p<文件名长度>　与-n参数相同，但此处的<文件名长度>包括了文件的路径。
- -w　指定输出时栏位的宽度。
- -V　显示版本信息。

实例

使用指令"which"查看指令"bash"的绝对路径，输入如下命令：

```
$ which bash
```

上面的指令执行后，输出信息如下所示：

```
/bin/bash                    #bash可执行程序的绝对路径
```

Linux cp命令

Linux cp命令主要用于复制文件或目录。

语法

```
cp [options] source dest
```

或

```
cp [options] source... directory
```

参数说明：

- -a：此选项通常在复制目录时使用，它保留链接、文件属性，并复制目录下的所有内容。其作用等于dpR参数组合。
- -d：复制时保留链接。这里所说的链接相当于Windows系统中的快捷方式。
- -f：覆盖已经存在的目标文件而不给出提示。
- -i：与-f选项相反，在覆盖目标文件之前给出提示，要求用户确认是否覆盖，回答"y"时目标文件将被覆盖。
- -p：除复制文件的内容外，还把修改时间和访问权限也复制到新文件中。
- -r：若给出的源文件是一个目录文件，此时将复制该目录下所有的子目录和文件。
- -l：不复制文件，只是生成链接文件。

实例

使用指令"cp"将当前目录"test/"下的所有文件复制到新目录"newtest"下，输入如下命令：

```
$ cp –r test/ newtest
```

注意：用户使用该指令复制目录时，必须使用参数"-r"或者"-R"。

Linux mcopy命令

Linux mcopy命令用来复制 MSDOS 格式文件到 Linux 中，或是由 Linux 中复制 MSDOS 文件到磁片上。

mcopy 可复制单一的文件到所指定的文件名称，或是复制数个文件到所指定的目录之中。来源与目的文件可为 MSDOS 或是 Linux 文件。

mcopy指令是一种mtools工具指令，可以在DOS系统中复制文件或者在DOS与Linux操作系统之间进行文件复制。

语法

```
mcopy [-bnmpQt/][源文件][目标文件或目录]
```

参数：

- b 批处理模式。这是为大量的文件复制进行最佳化的选项,但是当在复制文件过程中产生 crash 时，会有安全性的问题产生。/ 递回的复制。包含目录所含文件与其下所有子目录中的文件。
- -n 覆盖其他文件时，不需要进行确认而直接覆盖
- m 将源文件修改时间设置为目标文件的修改时间。
- p 将源文件的属性设置为目标文件的属性。
- Q 当复制多个文件产生错误时，尽快结束程序。
- t 转换为文本文件。
- o 在覆盖 MSDOS 文件时不会出现警示讯息。

实例

将 A 盘根目录中的 autoexec.bat 复制到目前工作目录之下：

```
mcopy a:autoexec.bat .
```

当复制的内容包括子目录和文件时，必须使用参数"-/"递归操作，因此该命令为：

```
mcopy -/ A:\*
```

执行该命令前先使用mdir 命令查看原来的目录结构，执行mcopy 之后可使用ls 命令查看复制之后Linux系统中的文件结构，结果如下：

```
cmd@cmd-desktop:~$ mdir -/ a:\* #查看A 盘中的文件
Volume in drive A has no label #加载信息
Volume Serial Number is 13D2~055C
Directory for A:/ #以下为目录信息
#文件名目录大小 修改时间
./TEST <DIR> 2009-09-23 16:59
AUTORUN.INF 265 2009-09-23 16:53
AUTORUN.BAT 43 2009-09-23 16:56
3 files 308 bytes #统计总大小
724 325 bytes free #剩余空间
cmd@cmd-desktop:~$ mcopy -/ A:\* #将A盘上的所有文件复制到当前工作目录
cmd@cmd-desktop:~$ls
TEST AUTORUN.INF AUTORUN.BAT #A盘中的内容复制到Linux文件系统结构中
```

Linux mshowfat命令

Linux mshowfat命令用于显示MS-DOS文件在FAT中的记录。

mshowfat为mtools工具指令，可显示MS-DOS文件在FAT中的记录编号。

语法

```
mshowfat [文件...]
```

参数说明：

[文件...]：执行操作的文件相对路径或者绝对路径

实例

使用指令mshowfat查看文件"autorun.bat"的FAT信息，输入如下命令：

```
$ mshowfat autorun.bat
```

以上命令执行后，文件"autorun.bat"的FAT相关信息将会被显示出来。

注意：执行操作的文件必须是DOS文件系统下的文件。

Linux rhmask命令

Linux rhmask命令用于对文件进行加密和解密操作。

执行rhmask指令可制作加密过的文件，方便用户在公开的网络上传输该文件，而不至于被任意盗用。

语法

```
rhmask [加密文件][输出文件] 或 rhmask [-d][加密文件][源文件][输出文件]
```

参数：

- -d 产生加密过的文件。

实例

使用指令"rhmask"将加密文件"code.txt"进行加密后，另存为输出文件"demo.txt"，输入如下命令：

```
$ rhmask code.txt demo.txt
```

以上命令执行后，文件"code.txt"将被加密后，另存为已经加密的文件"demo.txt"。

注意：该指令有两种语法，用户可以有选择性地进行使用即可。

Linux whereis命令

Linux whereis命令用于查找文件。

该指令会在特定目录中查找符合条件的文件。这些文件应属于原始代码、二进制文件，或是帮助文件。

该指令只能用于查找二进制文件、源代码文件和man手册页，一般文件的定位需使用locate命令。

语法

```
whereis [-bfmsu][-B <目录>...][-M <目录>...][-S <目录>...][文件...]
```

参数：

- -b 　只查找二进制文件。 *-B*<目录> 　只在设置的目录下查找二进制文件。 -f 　不显示文件名前的路径名称。 *-m* 　只查找说明文件。 -M<目录> 　只在设置的目录下查找说明文件。 *-s* 　只查找原始代码文件。 -S<目录> 　只在设置的目录下查找原始代码文件。* -u 　查找不包含指定类型的文件。

实例

使用指令"whereis"查看指令"bash"的位置，输入如下命令：

```
$ whereis bash
```

上面的指令执行后，输出信息如下所示：

```
bash:/bin/bash/etc/bash.bashrc/usr/share/man/man1/bash.1.gz
```

注意：以上输出信息从左至右分别为查询的程序名、bash路径、bash的man 手册页路径。

如果用户需要单独查询二进制文件或帮助文件，可使用如下命令：

```
$ whereis -b bash
$ whereis -m bash
```

输出信息如下：

```
$ whereis -b bash               #显示bash 命令的二进制程序
bash: /bin/bash /etc/bash.bashrc /usr/share/bash    # bash命令的二进制程序的地址
$ whereis -m bash               #显示bash 命令的帮助文件
bash: /usr/share/man/man1/bash.1.gz  #bash命令的帮助文件地址
```

Linux scp命令

Linux scp命令用于Linux之间复制文件和目录。

scp是 secure copy的缩写, scp是linux系统下基于ssh登陆进行安全的远程文件拷贝命令。

语法

```
scp [-1246BCpqrv] [-c cipher] [-F ssh_config] [-i identity_file]
[-l limit] [-o ssh_option] [-P port] [-S program]
[[user@]host1:]file1 [...] [[user@]host2:]file2
```

简易写法:

```
scp [可选参数] file_source file_target
```

参数说明：

- -1： 强制scp命令使用协议ssh1
- -2： 强制scp命令使用协议ssh2
- -4： 强制scp命令只使用IPv4寻址
- -6： 强制scp命令只使用IPv6寻址
- -B： 使用批处理模式（传输过程中不询问传输口令或短语）
- -C： 允许压缩。（将-C标志传递给ssh，从而打开压缩功能）
- -p：保留原文件的修改时间，访问时间和访问权限。
- -q： 不显示传输进度条。
- -r： 递归复制整个目录。
- -v：详细方式显示输出。scp和ssh(1)会显示出整个过程的调试信息。这些信息用于调试连接，验证和配置问题。
- -c cipher： 以cipher将数据传输进行加密，这个选项将直接传递给ssh。
- -F ssh_config： 指定一个替代的ssh配置文件，此参数直接传递给ssh。
- -i identity_file： 从指定文件中读取传输时使用的密钥文件，此参数直接传递给ssh。
- -l limit： 限定用户所能使用的带宽，以Kbit/s为单位。
- -o ssh_option： 如果习惯于使用ssh_config(5)中的参数传递方式，
- -P port：注意是大写的P, port是指定数据传输用到的端口号
- -S program： 指定加密传输时所使用的程序。此程序必须能够理解ssh(1)的选项。

实例

1、从本地复制到远程

命令格式：

```
scp local_file remote_username@remote_ip:remote_folder
或者
scp local_file remote_username@remote_ip:remote_file
或者
scp local_file remote_ip:remote_folder
或者
scp local_file remote_ip:remote_file
```

- 第1,2个指定了用户名，命令执行后需要再输入密码，第1个仅指定了远程的目录，文件名字不变，第2个指定了文件名；
- 第3,4个没有指定用户名，命令执行后需要输入用户名和密码，第3个仅指定了远程的目录，文件名字不变，第4个指定了文件名；

应用实例：

```
scp /home/space/music/1.mp3 root@www.w3cschool.cc:/home/root/others/music
scp /home/space/music/1.mp3 root@www.w3cschool.cc:/home/root/others/music/001.mp3
scp /home/space/music/1.mp3 www.w3cschool.cc:/home/root/others/music
scp /home/space/music/1.mp3 www.w3cschool.cc:/home/root/others/music/001.mp3
```

复制目录命令格式：

```
scp -r local_folder remote_username@remote_ip:remote_folder
或者
scp -r local_folder remote_ip:remote_folder
```

- 第1个指定了用户名，命令执行后需要再输入密码；
- 第2个没有指定用户名，命令执行后需要输入用户名和密码；

应用实例：

```
scp -r /home/space/music/ root@www.w3cschool.cc:/home/root/others/
scp -r /home/space/music/ www.w3cschool.cc:/home/root/others/
```

上面命令将本地 music 目录复制到远程 others 目录下。

2、从远程复制到本地

从远程复制到本地，只要将从本地复制到远程的命令的后2个参数调换顺序即可，如下实例

应用实例：

```
scp root@www.w3cschool.cc:/home/root/others/music /home/space/music/1.mp3
scp -r www.w3cschool.cc:/home/root/others/ /home/space/music/
```

说明

1.如果远程服务器防火墙有为scp命令设置了指定的端口，我们需要使用 -p 参数来设置命令的端口号，命令格式如下：

```
#scp命令使用端口号 4588
scp -p 4588 remote@www.w3cschool.cc:/usr/local/sin.sh /home/administrator
```

2.使用scp命令要确保使用的用户具有可读取远程服务器相应文件的权限，否则scp命令是无法起作用的。

Linux awk 命令

AWK是一种处理文本文件的语言，是一个强大的文本分析工具。

之所以叫AWK是因为其取了三位创始人 Alfred Aho，Peter Weinberger, 和 Brian Kernighan 的Family Name的首字符。

语法

```
awk [选项参数] 'script' var=value file(s)
或
awk [选项参数] -f scriptfile var=value file(s)
```

选项参数说明：

- -F fs or --field-separator fs
  指定输入文件折分隔符，fs是一个字符串或者是一个正则表达式，如-F:。
- -v var=value or --asign var=value
  赋值一个用户定义变量。
- -f scripfile or --file scriptfile
  从脚本文件中读取awk命令。
- -mf nnn and -mr nnn
  对nnn值设置内在限制，-mf选项限制分配给nnn的最大块数目；-mr选项限制记录的最大数目。这两个功能是Bell实验室版awk的扩展功能，在标准awk中不适用。
- -W compact or --compat, -W traditional or --traditional
  在兼容模式下运行awk。所以gawk的行为和标准的awk完全一样，所有的awk扩展都被忽略。
- -W copyleft or --copyleft, -W copyright or --copyright
  打印简短的版权信息。
- -W help or --help, -W usage or --usage
  打印全部awk选项和每个选项的简短说明。
- -W lint or --lint
  打印不能向传统unix平台移植的结构的警告。
- -W lint-old or --lint-old
  打印关于不能向传统unix平台移植的结构的警告。
- -W posix
  打开兼容模式。但有以下限制，不识别：/x、函数关键字、func、换码序列以及当fs是一个空格时，将新行作为一个域分隔符；操作符和=不能代替^和^=；fflush无效。
- -W re-interval or --re-inerval
  允许间隔正则表达式的使用，参考(grep中的Posix字符类)，如括号表达式[[:alpha:]]。

- -W source program-text or --source program-text
  使用program-text作为源代码，可与-f命令混用。
- -W version or --version
  打印bug报告信息的版本。

基本用法

log.txt文本内容如下：

```
2 this is a test
3 Are you like awk
This's a test
10 There are orange,apple,mongo
```

用法一：

```
awk '{[pattern] action}' {filenames}   # 行匹配语句 awk '' 只能用单引号
```

实例：

```
# 每行按空格或TAB分割，输出文本中的1、4项
 $ awk '{print $1,$4}' log.txt
 ---------------------------------------------
 2 a
 3 like
 This's
 10 orange,apple,mongo
 # 格式化输出
 $ awk '{printf "%-8s %-10s\n",$1,$4}' log.txt
 ---------------------------------------------
 2        a
 3        like
 This's
 10       orange,apple,mongo
```

用法二：

```
awk -F  #-F相当于内置变量FS, 指定分割字符
```

实例：

```
# 使用","分割
 $  awk -F, '{print $1,$2}'   log.txt
 ---------------------------------------------
 2 this is a test
 3 Are you like awk
 This's a test
 10 There are orange apple
 # 或者使用内建变量
 $ awk 'BEGIN{FS=","} {print $1,$2}'     log.txt
 ---------------------------------------------
 2 this is a test
 3 Are you like awk
 This's a test
 10 There are orange apple
 # 使用多个分隔符.先使用空格分割, 然后对分割结果再使用","分割
 $ awk -F '[ ,]'  '{print $1,$2,$5}'   log.txt
 ---------------------------------------------
 2 this test
 3 Are awk
 This's a
 10 There apple
```

用法三：

```
awk -v  # 设置变量
```

实例：

```
 $ awk -va=1 '{print $1,$1+a}' log.txt
 ---------------------------------------------
 2 3
 3 4
 This's 1
 10 11
 $ awk -va=1 -vb=s '{print $1,$1+a,$1b}' log.txt
 ---------------------------------------------
 2 3 2s
 3 4 3s
 This's 1 This'ss
 10 11 10s
```

用法四：

```
awk -f {awk脚本} {文件名}
```

实例：

```
 $ awk -f cal.awk log.txt
```

运算符

运算符	描述
= += -= = /= %= ^= *=	赋值
?:	C条件表达式
||	逻辑或
&&	逻辑与
~ ~!	匹配正则表达式和不匹配正则表达式
< <= > >= != ==	关系运算符
空格	连接
+ -	加，减
* / &	乘，除与求余
+ - !	一元加，减和逻辑非
^ *	求幂
++ --	增加或减少，作为前缀或后缀
$	字段引用
in	数组成员

过滤第一列大于2的行

```
$ awk '$1>2' log.txt    #命令
#输出
3 Are you like awk
This's a test
10 There are orange,apple,mongo
```

过滤第一列等于2的行

```
$ awk '$1==2 {print $1,$3}' log.txt    #命令
#输出
2 is
```

过滤第一列大于2并且第二列等于'Are'的行

```
$ awk '$1>2 && $2=="Are" {print $1,$2,$3}' log.txt    #命令
#输出
3 Are you
```

内建变量

变量	描述
$n	当前记录的第n个字段，字段间由FS分隔
$0	完整的输入记录
ARGC	命令行参数的数目
ARGIND	命令行中当前文件的位置(从0开始算)
ARGV	包含命令行参数的数组
CONVFMT	数字转换格式(默认值为%.6g)ENVIRON环境变量关联数组
ERRNO	最后一个系统错误的描述
FIELDWIDTHS	字段宽度列表(用空格键分隔)
FILENAME	当前文件名
FNR	同NR，但相对于当前文件
FS	字段分隔符(默认是任何空格)
IGNORECASE	如果为真，则进行忽略大小写的匹配
NF	当前记录中的字段数
NR	当前记录数
OFMT	数字的输出格式(默认值是%.6g)
OFS	输出字段分隔符(默认值是一个空格)
ORS	输出记录分隔符(默认值是一个换行符)
RLENGTH	由match函数所匹配的字符串的长度
RS	记录分隔符(默认是一个换行符)
RSTART	由match函数所匹配的字符串的第一个位置
SUBSEP	数组下标分隔符(默认值是/034)

```
$ awk 'BEGIN{printf "%4s %4s %4s %4s %4s %4s %4s %4s %4s\n","FILENAME","ARGC","FNR","FS",
FILENAME ARGC  FNR   FS   NF   NR  OFS  ORS   RS
---------------------------------------------
log.txt    2    1         5    1
log.txt    2    2         5    2
log.txt    2    3         3    3
log.txt    2    4         4    4
$ awk -F\' 'BEGIN{printf "%4s %4s %4s %4s %4s %4s %4s %4s %4s\n","FILENAME","ARGC","FNR",
FILENAME ARGC  FNR   FS   NF   NR  OFS  ORS   RS
---------------------------------------------
log.txt    2    1    '    1    1
log.txt    2    2    '    1    2
log.txt    2    3    '    2    3
log.txt    2    4    '    1    4
# 输出顺序号 NR, 匹配文本行号
$ awk '{print NR,FNR,$1,$2,$3}' log.txt
---------------------------------------------
1 1 2 this is
2 2 3 Are you
3 3 This's a test
4 4 10 There are
# 指定输出分割符
$  awk '{print $1,$2,$5}' OFS=" $ "  log.txt
---------------------------------------------
2 $ this $ test
3 $ Are $ awk
This's $ a $
10 $ There $
```

使用正则，字符串匹配

```
# 输出第二列包含 "th"，并打印第二列与第四列
$ awk '$2 ~ /th/ {print $2,$4}' log.txt
---------------------------------------------
this a
```

~ 表示模式开始。// 中是模式。

```
# 输出包含"re" 的行
$ awk '/re/ ' log.txt
---------------------------------------------
3 Are you like awk
10 There are orange,apple,mongo
```

忽略大小写

```
$ awk 'BEGIN{IGNORECASE=1} /this/' log.txt
---------------------------------------------
2 this is a test
This's a test
```

模式取反

```
$ awk '$2 !~ /th/ {print $2,$4}' log.txt
---------------------------------------------
Are like
a
There orange,apple,mongo
$ awk '!/th/ {print $2,$4}' log.txt
---------------------------------------------
Are like
a
There orange,apple,mongo
```

awk脚本

关于awk脚本，我们需要注意两个关键词BEGIN和END。

- BEGIN{ 这里面放的是执行前的语句 }
- END {这里面放的是处理完所有的行后要执行的语句 }
- {这里面放的是处理每一行时要执行的语句}

假设有这么一个文件（学生成绩表）：

```
$ cat score.txt
Marry   2143 78 84 77
Jack    2321 66 78 45
Tom     2122 48 77 71
Mike    2537 87 97 95
Bob     2415 40 57 62
```

我们的awk脚本如下：

```
$ cat cal.awk
#!/bin/awk -f
#运行前
BEGIN {
    math = 0
    english = 0
    computer = 0

    printf "NAME    NO.   MATH  ENGLISH  COMPUTER   TOTAL\n"
    printf "---------------------------------------------\n"
}
#运行中
{
    math+=$3
    english+=$4
    computer+=$5
    printf "%-6s %-6s %4d %8d %8d %8d\n", $1, $2, $3,$4,$5, $3+$4+$5
}
#运行后
END {
    printf "---------------------------------------------\n"
    printf "  TOTAL:%10d %8d %8d \n", math, english, computer
    printf "AVERAGE:%10.2f %8.2f %8.2f\n", math/NR, english/NR, computer/NR
}
```

我们来看一下执行结果：

```
$ awk -f cal.awk score.txt
NAME     NO.    MATH  ENGLISH  COMPUTER   TOTAL
---------------------------------------------
Marry  2143     78       84       77       239
Jack   2321     66       78       45       189
Tom    2122     48       77       71       196
Mike   2537     87       97       95       279
Bob    2415     40       57       62       159
---------------------------------------------
  TOTAL:       319      393      350
AVERAGE:     63.80    78.60    70.00
```

另外一些实例

AWK的hello world程序为：

```
BEGIN { print "Hello, world!" }
```

计算文件大小

```
$ ls -l *.txt | awk '{sum+=$6} END {print sum}'
--------------------------------------------------
666581
```

从文件中找出长度大于80的行

```
awk 'lenght>80' log.txt
```

打印九九乘法表

```
seq 9 | sed 'H;g' | awk -v RS='' '{for(i=1;i<=NF;i++)printf("%dx%d=%d%s", i, NR, i*NR, i=
```

更多详细内容可以查看 AWK 官方手册：http://www.gnu.org/software/gawk/manual/gawk.html

Linux命令大全 - 文档编辑

col	colrm	comm	csplit
ed	egrep	ex	fgrep
fmt	fold	grep	ispell
jed	joe	join	look
mtype	pico	rgrep	sed
sort	spell	tr	expr
uniq	wc		

Linux col命令

Linux col命令用于过滤控制字符。

在许多UNIX说明文件里，都有RLF控制字符。当我们运用shell特殊字符">"和">>"，把说明文件的内容输出成纯文本文件时，控制字符会变成乱码，col指令则能有效滤除这些控制字符。

语法

```
col [-bfx][-l<缓冲区列数>]
```

参数：

- -b 过滤掉所有的控制字符，包括RLF和HRLF。
- -f 滤除RLF字符，但允许将HRLF字符呈现出来。
- -x 以多个空格字符来表示跳格字符。
- -l<缓冲区列数> 预设的内存缓冲区有128列，您可以自行指定缓冲区的大小。

实例

下面以 man 命令帮助文档为例，讲解col 命令的使用。

将man 命令的帮助文档保存为man_help，使用-b 参数过滤所有控制字符。在终端中使用如下命令：

```
man man | col-b > man_help
```

注:其中"|"用于建立管道，把man命令的输出结果转为col命令的输入数据。

Linux colrm命令

Linux colrm命令用于滤掉指定的行。

colrm指令从标准输入设备读取书记，转而输出到标准输出设备。如果不加任何参数，则该指令不会过滤任何一行。

语法

```
colrm [开始行数编号<结束行数编号>]</p>

<p><b>参数说明：</b></p>
<ul>
<li>开始行数编号： 指定要删除的列的起始编号。</li>
<li>结束行数编号： 指定要删除的列的结束编号，有时候这个参数可以省略。</li>
</ul>
<h3>实例</h3>
<p>不带任何参数时该命令不会删除任何列：</p>
<pre>
colrm
```

按回车键后，光标将在第一行闪烁，等待标准输入，此时输入字符，如"Hello Linux！"，再按回车键后第二行将出现与第一行相同内容，此时按Ctrl+C组合键可以退出。终端中显示的内容如下所示：

```
cmd@hdd-desktop:~$ colrm
Hello Linux! #输入Hello Linux！字符串
Hello Linux! #输出刚才输入的字符串Hello Linux！
```

如想要删除第4 列之后的所有内容，可以使用如下命令：

```
colrm 4
```

类似于上例，此时标准输入等待输入，用户输入字符串按回车键后，将输出如下结果：

```
cmd@hdd-desktop:~$ colrm 4
Hello Linux! #输入Hello Linux！字符串
Hel #输出删除了第4列以后所有内容的字符串
```

删除指定列的内容。如删除第4列到第6列的内容，可使用如下命令：

```
colrm 4 6
```

输出的结果如下：

```
cmd@hdd-desktop:~$ colrm 4 6
Hello Linux! #输入Hello Linux！字符串
HelLinux! #输出删除了从第4列到第6列字符的字符串
```

Linux comm命令

Linux comm命令用于比较两个已排过序的文件。

这项指令会一列列地比较两个已排序文件的差异，并将其结果显示出来，如果没有指定任何参数，则会把结果分成3行显示：第1行仅是在第1个文件中出现过的列，第2行是仅在第2个文件中出现过的列，第3行则是在第1与第2个文件里都出现过的列。若给予的文件名称为"-"，则comm指令会从标准输入设备读取数据。

语法

```
comm [-123][--help][--version][第1个文件][第2个文件]
```

参数：

- -1 不显示只在第1个文件里出现过的列。
- -2 不显示只在第2个文件里出现过的列。
- -3 不显示只在第1和第2个文件里出现过的列。
- --help 在线帮助。
- --version 显示版本信息。

实例

aaa.txt 与 bbb.txt 的文件内容如下：

```
[root@localhost text]# cat aaa.txt
aaa
bbb
ccc
ddd
eee
111
222
[root@localhost text]# cat bbb.txt
bbb
ccc
aaa
hhh
ttt
jjj
```

```
<p>执行 comm 命令输出结果如下：</p>
[root@localhost text]# comm aaa.txt bbb.txt
aaa
                bbb
                ccc
        aaa
ddd
eee
111
222
        hhh
        ttt
        jjj
第一列  第二列  第三列
```

输出的第一列只包含在aaa.txt中出现的行，第二列包含在bbb.txt中出现的行，第三列包含在aaa.txt和bbb.txt中相同的行。各列是以制表符（\t）作为定界符。

Linux csplit命令

Linux csplit命令用于分割文件。

将文件依照指定的范本样式予以切割后，分别保存成名称为xx00,xx01,xx02...的文件。若给予的文件名称为"-"，则csplit指令会从标准输入设备读取数据。

语法

```
csplit [-kqsz][-b<输出格式>][-f<输出字首字符串>]
[-n<输出文件名位数>][--help][--version][文件][范本样式...]
```

参数：

- -b<输出格式>或--suffix-format=<输出格式> 预设的输出格式其文件名称为xx00,xx01...等，您可以通过改变<输出格式>来改变输出的文件名。
- -f<输出字首字符串>或--prefix=<输出字首字符串> 预设的输出字首字符串其文件名为xx00,xx01...等，如果你指定输出字首字符串为"hello"，则输出的文件名称会变成hello00,hello01...等。
- -k或--keep-files 保留文件，就算发生错误或中断执行，也不能删除已经输出保存的文件。
- -n<输出文件名位数>或--digits=<输出文件名位数> 预设的输出文件名位数其文件名称为xx00,xx01...等，如果你指定输出文件名位数为"3"，则输出的文件名称会变成xx000,xx001...等。
- -q或-s或--quiet或--silent 不显示指令执行过程。
- -z或--elide-empty-files 删除长度为0 Byte文件。
- --help 在线帮助。
- --version 显示版本信息。

实例

将文本文件testfile以第 2 行为分界点切割成两份，使用如下命令：

```
csplit testfile 2
```

testfile文件中的内容如下：

```
$ cat testfile #查看testfile 文件内容
hello Linux!
Linux is a free Unix-type operating system.
This is a Linux testfile!
Linux
```

使用csplit命令，输出结果如下：

```
$ csplit testfile 2
13 #xx00文件字符个数
76 #xx01文件字符个数
```

其中第1 行是第一个文件xx00的字符个数，同样，第2 行为第二个文件xx01的字符个数。同时，在testfile 的同目录下将生成两个文件，文件名分别为xx00、xx01，xx00 中的内容为：

```
$ cat xx00 #查看分割后的xx00文件内容
hello Linux! #testfile文件第1行的内容
```

xx01 中的内容为：

```
$ cat xx01 #查看分割后的xx01文件内容
Linux is a free Unix-type operating system. #testfile文件第2行以后的内容
This is a Linux testfile!
Linux
```

Linux ed命令

Linux ed命令是文本编辑器，用于文本编辑。

ed是Linux中功能最简单的文本编辑程序，一次仅能编辑一行而非全屏幕方式的操作。

ed命令并不是一个常用的命令，一般使用比较多的是vi 指令。但ed文本编辑器对于编辑大文件或对于在shell脚本程序中进行文本编辑很有用。

语法

```
ed [-][-Gs][-p<字符串>][--help][--version][文件]
```

参数：

- -G或--traditional 提供回兼容的功能。
- -p<字符串> 指定ed在command mode的提示字符。
- -s,-,--quiet或--silent 不执行开启文件时的检查功能。
- --help 显示帮助。
- --version 显示版本信息。

实例

以下是一个 Linux ed 完整实例解析：

```
$ ed              <- 激活 ed 命令
a                 <- 告诉 ed 我要编辑新文件
My name is Titan. <- 输入第一行内容
And I love Perl very much. <- 输入第二行内容
.                 <- 返回 ed 的命令行状态
i                 <- 告诉 ed 我要在最后一行之前插入内容
I am 24\.          <- 将“I am 24.”插入“My name is Titan.”和“And I love Perl very much.”之间
.                 <- 返回 ed 的命令行状态
c                 <- 告诉 ed 我要替换最后一行输入内容
I am 24 years old. <- 将“I am 24.”替换成“I am 24 years old.”（注意：这里替换的是最后输的内容）
.                 <- 返回 ed 的命令行状态
w readme.text     <- 将文件命名为“readme.text”并保存（注意：如果是编辑已经存在的文件，只需要敲入 w .
q                 <- 完全退出 ed 编辑器
```

这是文件的内容是：

```
$ cat readme.text
My name is Titan.
I am 24 years old.
And I love Perl vrey much.
```

Linux egrep命令

Linux egrep命令用于在文件内查找指定的字符串。

egrep执行效果与"grep-E"相似，使用的语法及参数可参照grep指令，与grep的不同点在于解读字符串的方法。

egrep是用extended regular expression语法来解读的，而grep则用basic regular expression语法解读，extended regular expression比basic regular expression的表达更规范。

语法

```
egrep [范本模式] [文件或目录]
```

参数说明：

- [范本模式] ：查找的字符串规则。
- [文件或目录] ：查找的目标文件或目录。

实例

显示文件中符合条件的字符。例如，查找当前目录下所有文件中包含字符串"Linux"的文件，可以使用如下命令：

```
egrep Linux *
```

结果如下所示：

```
$ egrep Linux * #查找当前目录下包含字符串“Linux”的文件
testfile:hello Linux! #以下五行为testfile 中包含Linux字符的行
testfile:Linux is a free Unix-type operating system.
testfile:This is a Linux testfile!
testfile:Linux
testfile:Linux
testfile1:helLinux! #以下两行为testfile1中含Linux字符的行
testfile1:This a Linux testfile!
#以下两行为testfile_2 中包含Linux字符的行
testfile_2:Linux is a free unix-type opterating system.
testfile_2:Linux test
xx00:hello Linux! #xx00包含Linux字符的行
xx01:Linux is a free Unix-type operating system. #以下三行为xx01包含Linux字符的行
xx01:This is a Linux testfile!
xx01:Linux
```

Linux ex命令

Linux ex命令用于在Ex模式下启动vim文本编辑器。

ex执行效果如同vi -E，使用语法及参数可参照vi指令，如要从Ex模式回到普通模式，则在vim中输入":vi"或":visual"指令即可。

语法

```
ex [选项][参数]
```

参数说明：

- -b：使用二进制模式编辑文件
- -c 指令：编辑完第一个文件后执行指定的指令
- -d ：编辑多个文件时，显示差异部分
- -m ：不允许修改文件
- -n ：不使用缓存
- -oN：其中 N 为数字
- -r ：列出缓存，并显示恢复信息
- -R ：以只读的方式打开文件
- -s ：不显示任何错误信息
- -V ：显示指令的详细执行过程
- --help ：显示帮助信息
- --version ：显示版本信息

实例

在ex 指令后输入文件名按回车键后，即可进入ex 编辑模式，如编辑testfile文件，使用的命令格式如下：

```
ex testfile
```

输出的信息如下：

```
"testfile" 5L, 95C
```

"testfile"表示文件名，5L表示5 行，95 表示字节数

进入ex 模式。输入"visual"回到正常模式

它的操作与vim 中是一样的，此时如果在":"后输入"visual"后按回车键，将进入到vi 指令全屏界面；如果输入"q"，则退出编辑器。

Linux fgrep命令

本指令相当于执行grep指令加上参数"-F"，详见grep命令说明。

Linux fgrep命令用于查找文件里符合条件的字符串。

语法

```
fgrep [范本样式][文件或目录...]
```

实例

具体使用实例请参考grep命令。

Linux fmt命令

Linux fmt命令用于编排文本文件。

fmt指令会从指定的文件里读取内容，将其依照指定格式重新编排后，输出到标准输出设备。若指定的文件名为"-"，则fmt指令会从标准输入设备读取数据。

语法

```
fmt [-cstu][-p<列起始字符串>][-w<每列字符数>][--help][--version][文件...]
```

参数说明：

- -c或--crown-margin 每段前两列缩排。
- -p<列起始字符串>或-prefix=<列起始字符串> 仅合并含有指定字符串的列，通常运用在程序语言的注解方面。
- -s或--split-only 只拆开字数超出每列字符数的列，但不合并字数不足每列字符数的列。
- -t或--tagged-paragraph 每列前两列缩排，但第1列和第2列的缩排格式不同。
- -u或--uniform-spacing 每个字符之间都以一个空格字符间隔，每个句子之间则两个空格字符分隔。
- -w<每列字符数>或--width=<每列字符数>或-<每列字符数> 设置每列的最大字符数。
- --help 在线帮助。
- --version 显示版本信息。

实例

重排指定文件。如文件testfile共5 行文字，可以通过命令对该文件格式进行重排，其命令为：

```
fmt testfile
```

输出结果如下：

```
$ fmt testfile #重排testfile 文件
hello Linux! Linux is a free Unix-type operating system. This is a
Linux testfile! Linux Linux
```

将文件testfile重新排成85 个字符一行，并在标准输出设备上输出，其命令应该为：

```
fmt -w 85 testfile
```

为了对比，先使用cat 命令查看文件内容：

```
$ cat testfile #查看testfile 文件的内容
hello Linux!
Linux is a free Unix-type operating system.
This is a Linux testfile!
Linux
Linux
```

使用fmt命令重排之后，输出结果如下：

```
$ fmt -w 85 testfile #指定重排宽度为85个字符
hello Linux! Linux is a free Unix-type operating system. This is a Linux testfile!
Linux Linux
```

Linux fold命令

Linux fold命令用于限制文件列宽。

fold指令会从指定的文件里读取内容，将超过限定列宽的列加入增列字符后，输出到标准输出设备。若不指定任何文件名称，或是所给予的文件名为"-"，则fold指令会从标准输入设备读取数据。

语法

```
fold [-bs][-w<每列行数>][--help][--version][文件...]
```

参数：

- -b或--bytes 以Byte为单位计算列宽，而非采用行数编号为单位。
- -s或--spaces 以空格字符作为换列点。
- -w<每列行数>或--width<每列行数> 设置每列的最大行数。
- --help 在线帮助。
- --version 显示版本信息。

实例

将一个名为testfile 的文件的行折叠成宽度为30，可使用如下命令：

```
fold -w 30 testfile
```

为了对比，先将testfile文件输出如下：

```
$ cat testfile #查看testfile 中的内容
Linux networks are becoming more and more common, but
security is often an overlooked
issue. Unfortunately, in today’s environment all networks
are potential hacker targets,
from top-secret military research networks to small home LANs.
Linux Network Security focuses on securing Linux in a
networked environment, where the
security of the entire network needs to be considered
rather than just isolated machines.
It uses a mix of theory and practical techniques to
teach administrators how to install and
use security applications, as well as how the
applications work and why they are necessary.
```

然后使用fold命令折叠显示：

```
$ fold -w 30 testfile #行折叠成宽度为30，显示testfile 文件
Linux networks are becoming mo
re and more common, but securi
ty is often an overlooked issu
e. Unfortunately, in today’s
environment all networks are
potential hacker targets, from
top-secret military research
networks to small home LANs.
Linux Network Security focuses
on securing Linux in a networ
ked environment, where the sec
urity of the entire network ne
eds to be considered rather th
an just isolated machines. It
uses a mix of theory and pract
ical techniques to teach admin
istrators how to install and u
se security applications, as w
ell as how the applications wo
rk and why they are necessary
```

Linux grep命令

Linux grep命令用于查找文件里符合条件的字符串。

grep指令用于查找内容包含指定的范本样式的文件，如果发现某文件的内容符合所指定的范本样式，预设grep指令会把含有范本样式的那一列显示出来。若不指定任何文件名称，或是所给予的文件名为"-"，则grep指令会从标准输入设备读取数据。

语法

```
grep [-abcEFGhHilLnqrsvVwxy][-A<显示列数>][-B<显示列数>][-C<显示列数>][-d<进行动作>][-e<范本样式
```

参数：

- -a或--text 不要忽略二进制的数据。
- -A<显示列数>或--after-context=<显示列数> 除了显示符合范本样式的那一列之外，并显示该列之后的内容。
- -b或--byte-offset 在显示符合范本样式的那一列之前，标示出该列第一个字符的位编号。
- -B<显示列数>或--before-context=<显示列数> 除了显示符合范本样式的那一列之外，并显示该列之前的内容。
- -c或--count 计算符合范本样式的列数。
- -C<显示列数>或--context=<显示列数>或-<显示列数> 除了显示符合范本样式的那一列之外，并显示该列之前后的内容。
- -d<进行动作>或--directories=<进行动作> 当指定要查找的是目录而非文件时，必须使用这项参数，否则grep指令将回报信息并停止动作。
- -e<范本样式>或--regexp=<范本样式> 指定字符串做为查找文件内容的范本样式。
- -E或--extended-regexp 将范本样式为延伸的普通表示法来使用。
- -f<范本文件>或--file=<范本文件> 指定范本文件，其内容含有一个或多个范本样式，让grep查找符合范本条件的文件内容，格式为每列一个范本样式。
- -F或--fixed-regexp 将范本样式视为固定字符串的列表。
- -G或--basic-regexp 将范本样式视为普通的表示法来使用。
- -h或--no-filename 在显示符合范本样式的那一列之前，不标示该列所属的文件名称。
- -H或--with-filename 在显示符合范本样式的那一列之前，表示该列所属的文件名称。
- -i或--ignore-case 忽略字符大小写的差别。
- -l或--file-with-matches 列出文件内容符合指定的范本样式的文件名称。
- -L或--files-without-match 列出文件内容不符合指定的范本样式的文件名称。
- -n或--line-number 在显示符合范本样式的那一列之前，标示出该列的列数编号。
- -q或--quiet或--silent 不显示任何信息。

- -r或--recursive 此参数的效果和指定"-d recurse"参数相同。
- -s或--no-messages 不显示错误信息。
- -v或--revert-match 反转查找。
- -V或--version 显示版本信息。
- -w或--word-regexp 只显示全字符合的列。
- -x或--line-regexp 只显示全列符合的列。
- -y 此参数的效果和指定"-i"参数相同。
- --help 在线帮助。

实例

1、在当前目录中，查找后缀有"test"字样的文件中包含"test"字符串的文件，并打印出该字符串的行。此时，可以使用如下命令：

```
grep test *file
```

结果如下所示：

```
$ grep test test* #查找后缀有“test”的文件包含“test”字符串的文件
testfile1:This a Linux testfile! #列出testfile1 文件中包含test字符的行
testfile_2:This is a linux testfile! #列出testfile_2 文件中包含test字符的行
testfile_2:Linux test #列出testfile_2 文件中包含test字符的行
```

2、以递归的方式查找符合条件的文件。例如，查找指定目录/etc/acpi 及其子目录（如果存在子目录的话）下所有文件中包含字符串"update"的文件，并打印出该字符串所在行的内容，使用的命令为：

```
grep -r update /etc/acpi
```

输出结果如下：

```
$ grep-r update /etc/acpi #以递归的方式查找“etc/acpi”
#下包含“update”的文件
/etc/acpi/ac.d/85-anacron.sh:# (Things like the slocate updatedb cause a lot of IO.)
Rather than
/etc/acpi/resume.d/85-anacron.sh:# (Things like the slocate updatedb cause a lot of
IO.) Rather than
/etc/acpi/events/thinkpad-cmos:action=/usr/sbin/thinkpad-keys--update
```

3、反向查找。前面各个例子是查找并打印出符合条件的行，通过"-v"参数可以打印出不符合条件行的内容。

查找文件名中包含test 的文件中不包含test 的行，此时，使用的命令为：

```
grep -v test*
```

结果如下所示：

```
$ grep-v test* #查找文件名中包含test 的文件中不包含test 的行
testfile1:helLinux!
testfile1:Linis a free Unix-type operating system.
testfile1:Lin
testfile_1:HELLO LINUX!
testfile_1:LINUX IS A FREE UNIX-TYPE OPTERATING SYSTEM.
testfile_1:THIS IS A LINUX TESTFILE!
testfile_2:HELLO LINUX!
testfile_2:Linux is a free unix-type opterating system.
```

Linux ispell命令

Linux ispell命令用于拼写检查程序。

ispell预设会使用/usr/lib/ispell/english.hash字典文件来检查文本文件。若在检查的文件中找到字典没有的词汇，ispell会建议使用的词汇，或是让你将新的词汇加入个人字典。

语法

```
ispell [-aAbBClmMnNPStVx][-d<字典文件>][-L<行数>][-p<字典文件>][-w<非字母字符>][-W<字符串长度>][
```

参数：

- -a 当其他程序输出送到ispell时，必须使用此参数。
- -A 读取到"&Include File&"字符串时，就去检查字符串后所指定文件的内容。
- -b 产生备份文件，文件名为.bak。
- -B 检查连字错误。
- -C 不检查连字错误。
- -d<字典文件> 指定字典文件。
- -l 从标准输入设备读取字符串，结束后显示拼错的词汇。
- -L<行数> 指定内文显示的行数。
- -m 自动考虑字尾的变化。
- -M 进入ispell后，在画面下方显示指令的按键。
- -n 检查的文件为noff或troff的格式。
- -N 进入ispell后，在画面下方不显示指令的按键。
- -p<字典文件> 指定个人字典文件。
- -P 不考虑字尾变化的情形。
- -S 不排序建议取代的词汇。
- -t 检查的文件为TeX或LaTeX的格式。
- -V 非ANSI标准的字符会以"M-^"的方式来显示。
- -w<非字母字符> 检查时，特别挑出含有指定的字符。
- -W<字符串长度> 不检查指定长度的词汇。
- -x 不要产生备份文件。

实例

检查文件的拼写。例如，检查testfile文件，可使用如下命令：

```
ispell testfile
```

如果文件中出现可疑词汇，则第一个出现的可疑词汇以高亮显示，并在屏幕下方给出词汇的修改意见，以及ispell的操作命令。如下所示：

```
netwrks File: testfile
Linux netwrks are becoming more and more common, but security is often an overlooked
issue. Unfortunately
0: networks
[SP] <number> R)epl A)ccept I)nsert L)ookup U)ncap Q)uit e(X)it or ? for help
```

本例中，检查出netwrks 错误，并提示纠正信息，此时输入"0"，即使用networks 来纠正错误，同时继续显示下一个错误，直到所有的错误显示完毕。

通过以上实例我们可以发现，文件testfile中有拼写错误，对该文件进行修改后需备份文件。此时使用如下命令：

```
ispell-b testfile    #检查拼写错误的同时，备份文件
```

如果文件已经无拼写错误，则不显示任何信息，通过ls命令我们也可以查看到当前文件目录下产生了文件testfile的备份文件testfile.bak。查看结果如下所示：

```
$ ls #以列表的形式查看当前目录下的文件
examples.desktop testfile_1 testfile.bak xx01 模板图片 音乐
testfile testfile1 testfile_2 xx00 公共的视频文档桌面
```

其中，testfile.bak 文件就是刚才命令生成的备份文件，内容与原来的testfile 文件内容是一样的。

Linux jed命令

Linux jed命令用于编辑文本文件。

Jed是以Slang所写成的程序，适合用来编辑程序原始代码。

语法

```
jed [-2n][-batch][-f<函数>][-g<行数>][-i<文件>][-I<文件>][-s<字符串>][文件]
```

参数：

- -2 显示上下两个编辑区。
- -batch 以批处理模式来执行。
- -f<函数> 执行Slang函数。
- -g<行数> 移到缓冲区中指定的行数。
- -i<文件> 将指定的文件载入缓冲区。
- -i<文件> 载入Slang原始代码文件。
- -n 不要载入jed.rc配置文件。
- -s<字符串> 查找并移到指定的字符串。

实例

jed主要用于编辑程序的源码，编辑源码时将以彩色高亮的方式显示程序的语法。例如使用jed编辑一个C语言的源代码文件，可使用如下命令：

```
jed main.c       #用jed编辑器打开main.c 文件
```

输出结果如下：

```
F10 key ==> File Edit Mode Search Buffers Windows System Help #编辑器菜单
/*-*- linux-c-*-*/ #编辑区
#include <linux/mm.h>
#include <linux/sysctl.h>
#include <linux/nsproxy.h>
static struct list_head *
net_ctl_header_lookup(struct ctl_table_root *root, struct nsproxy *namespaces)
{
return &namespaces->net_ns->sysctl_table_headers;
}
static struct ctl_table_root net_sysctl_root = {
.lookup = net_ctl_header_lookup,
};
static int sysctl_net_init(struct net *net)
{
INIT_LIST_HEAD(&net->sysctl_table_headers);
return 0;
}
-----+(Jed 0.99.18U) Emacs: main.c (C) All 6:06pm-----------------------------
#从左到右分别为jed版本编号、当前是模拟emacs编辑器、打开的文件名、现在的时间
loading /usr/share/jed/lib/modeinfo.slc
```

Linux joe命令

Linux joe命令用于编辑文本文件。

Joe是一个功能强大的全屏幕文本编辑程序。操作的复杂度要比Pico高一点，但是功能较为齐全。Joe一次可开启多个文件，每个文件各放在一个编辑区内，并可在文件之间执行剪贴的动作。

语法

```
joe [-asis][-beep][-csmode][-dopadding][-exask][-force][-help][-keepup][-lightoff][-arkin
```

参数：

- 以下为程序参数 *-asis* 字符码超过*127*的字符不做任何处理。 -backpath<目录> 指定备份文件的目录。* -beep 编辑时，若有错误即发出哔声。
- -columns<栏位> 设置栏数。
- -csmode 可执行连续查找模式。
- -dopadding 是程序跟tty间存在缓冲区。
- -exask 在程序中，执行"Ctrl+k+x"时，会先确认是否要保存文件。
- -force 强制在最后一行的结尾处加上换行符号。
- -help 执行程序时一并显示帮助。
- -keepup 在进入程序后，画面上方为状态列。
- -lightoff 选取的区块在执行完区块命令后，就会回复成原来的状态。
- -lines<行数> 设置行数。
- -marking 在选取区块时，反白区块会随着光标移动。
- -mid 当光标移出画面时，即自动卷页，使光标回到中央。
- -nobackups 不建立备份文件。
- -nonotice 程序执行时，不显示版权信息。
- -nosta 程序执行时，不显示状态列。
- -noxon 尝试取消"Ctrl+s"与"Ctrl+q"键的功能。
- -orphan 若同时开启一个以上的文件，则其他文件会置于独立的缓冲区，而不会另外开启编辑区。
- -pg<行数> 按"PageUp"或"PageDown"换页时，所要保留前一页的行数。
- -skiptop<行数> 不使用屏幕上方指定的行数。
- 以下为文件参数
- +<行数> 指定开启文件时，光标所在的行数。

- -autoindent 自动缩排。
- -crlf 在换行时，使用CR-LF字符。
- -indentc<缩排字符> 执行缩排时，实际插入的字符。
- -istep<缩排字符数> 每次执行缩排时，所移动的缩排字符数。
- -keymap<按键配置文件> 使用不同的按键配置文件。
- -linums 在每行前面加上行号。
- -lmargin<栏数> 设置左侧边界。
- -overwrite 设置覆盖模式。
- -rmargin<栏数> 设置右侧边界。
- -tab<栏数> 设置tab的宽度。
- -rdonly 以只读的方式开启文件-wordwrap编辑时若超过右侧边界，则自动换行。

实例

利用joe命令编辑文本文件。例如利用joe编辑C 语言源代码main.c，使用如下命令：

```
joe main.c
```

与jed类似，joe编辑器中C语言的语法也以彩色的方式显示。效果如下：

```
I A main.c (c) Row 1 Col 1 12:28 Ctrl-K H for help
#上排从左至右分别为打开的文件名、光标所在行列数、现在时间、显示操作说明
/*-*- linux-c-*-*/ #编辑区
#include <linux/mm.h>
#include <linux/sysctl.h>
#include <linux/nsproxy.h>
static struct list_head *
net_ctl_header_lookup(struct ctl_table_root *root, struct nsproxy *namespaces)
{
return &namespaces->net_ns->sysctl_table_headers;
}
static struct ctl_table_root net_sysctl_root = {
.lookup = net_ctl_header_lookup,
};
static int sysctl_net_init(struct net *net)
{
INIT_LIST_HEAD(&net->sysctl_table_headers);
return 0;
}
** Joe's Own Editor v3.5 ** (utf-8) ** Copyright . 2006 ** #joe编辑区的版本及版权信息
```

joe编辑器有一些常用的组合键，例如可以通过Ctrl+K+H 寻求联机帮助，首先按Ctrl+K组合键，再输入字母H，即可调出帮助菜单，通过该帮助信息可以方便地获知如何对joe 编辑器进行操作。

Linux join命令

Linux join命令用于将两个文件中，指定栏位内容相同的行连接起来。

找出两个文件中，指定栏位内容相同的行，并加以合并，再输出到标准输出设备。

语法

```
join [-i][-a<1或2>][-e<字符串>][-o<格式>][-t<字符>][-v<1或2>][-1<栏位>][-2<栏位>][--help][--v
```

参数：

- -a<1或2> 除了显示原来的输出内容之外，还显示指令文件中没有相同栏位的行。
- -e<字符串> 若[文件1]与[文件2]中找不到指定的栏位，则在输出中填入选项中的字符串。
- -i或--igore-case 比较栏位内容时，忽略大小写的差异。
- -o<格式> 按照指定的格式来显示结果。
- -t<字符> 使用栏位的分隔字符。
- -v<1或2> 跟-a相同，但是只显示文件中没有相同栏位的行。
- -1<栏位> 连接[文件1]指定的栏位。
- -2<栏位> 连接[文件2]指定的栏位。
- --help 显示帮助。
- --version 显示版本信息。

实例

连接两个文件。

为了清楚地了解join命令，首先通过cat命令显示文件testfile_1和 testfile_2 的内容。

然后以默认的方式比较两个文件，将两个文件中指定字段的内容相同的行连接起来，在终端中输入命令：

```
join testfile_1 testfile_2
```

首先查看testfile_1、testfile_2 中的文件内容：

```
$ cat testfile_1 #testfile_1文件中的内容
Hello 95 #例如，本例中第一列为姓名，第二列为数额
Linux 85
test 30
cmd@hdd-desktop:~$ cat testfile_2 #testfile_2文件中的内容
Hello 2005 #例如，本例中第一列为姓名，第二列为年份
Linux 2009
test 2006
```

然后使用join命令，将两个文件连接，结果如下：

```
$ join testfile_1 testfile_2 #连接testfile_1、testfile_2中的内容
Hello 95 2005 #连接后显示的内容
Linux 85 2009
test 30 2006
```

文件1与文件2的位置对输出到标准输出的结果是有影响的。例如将命令中的两个文件互换，即输入如下命令：

```
join testfile_2 testfile_1
```

最终在标准输出的输出结果将发生变化，如下所示：

```
$ join testfile_2 testfile_1 #改变文件顺序连接两个文件
Hello 2005 95 #连接后显示的内容
Linux 2009 85
test 2006 30
```

Linux look命令

Linux look命令用于查询单词。

look指令用于英文单字的查询。您仅需给予它欲查询的字首字符串，它会显示所有开头字符串符合该条件的单字。

语法

```
look [-adf][-t<字尾字符串>][字首字符串][字典文件]
```

参数说明：

- -a 使用另一个字典文件web2，该文件也位于/usr/dict目录下。
- -d 只对比英文字母和数字，其余一概忽略不予比对。
- -f 忽略字符大小写差别。
- -t<字尾字符串> 设置字尾字符串。

实例

为了查找在testfile文件中以字母L开头的所有的行，可以输入如下命令：

```
look L testfile
```

原文件testfile中的内容如下：

```
$ cat testfile #查看testfile 文件内容
HELLO LINUX!
Linux is a free unix-type opterating system.
This is a linux testfile!
Linux test
```

在testfile文件中使用look命令查找以"L"开头的单词，结果如下：

```
$ look L testfile                             #查找以“L”开头的单词
Linux is a free unix-type opterating system.  #第二行以“L”开头，列出全句
Linux test                                    #第四行以“L”开头，列出全句
```

Linux mtype命令

mtype为mtools工具指令，模拟MS-DOS的type指令，可显示MS-DOS文件的内容。

语法

```
mtype [-st][文件]
```

参数说明：

- -s 去除8位字符码集的第一个位，使它兼容于7位的ASCII。
- -t 将MS-DOS文本文件中的"换行+光标移至行首"字符转换成Linux的换行字符。

实例

打开名为dos.txt 的MS-DOS文件可使用如下命令：

```
mtype dos.txt         #打开MS-DOS 文件
```

显示结果如下：

```
$ mtype dos.txt #打开MS-DOS 文件
Linux networks are becoming more and more common, but security is often an overlooked
issue. Unfortunately, in today’s environment all networks are potential hacker targets,
from top-secret military research networks to small home LANs.
Linux Network Securty focuses on securing Linux in a networked environment, where the
security of the entire network needs to be considered rather than just isolated machines.
It uses a mix of theory and practicl techniques to teach administrators how to install an
use security applications, as well as how the applcations work and why they are necessary
```

Linux pico命令

Linux pico命令用于编辑文字文件。

pico是个简单易用、以显示导向为主的文字编辑程序，它伴随着处理电子邮件和新闻组的程序pine而来。

语法

```
pico [-bdefghjkmqtvwxz][-n<间隔秒数>][-o<工作目录>][-r<编辑页宽>][-s<拼字检查器>][+<列数编号>][文
```

参数说明：

- -b 开启置换的功能。
- -d 开启删除的功能。
- -e 使用完整的文件名称。
- -f 支持键盘上的F1、F2...等功能键。
- -g 显示光标。
- -h 在线帮助。
- -j 开启切换的功能。
- -k 预设pico在使用剪下命令时，会把光标所在的列的内容全部删除。
- -m 开启鼠标支持的功能，您可用鼠标点选命令列表。
- -n<间隔秒数> 设置多久检查一次新邮件。
- -o<工作目录> 设置工作目录。
- -q 忽略预设值。
- -r<编辑页宽> 设置编辑文件的页宽。
- -s<拼字检查器> 另外指定拼字检查器。
- -t 启动工具模式。
- -v 启动阅读模式，用户只能观看，无法编辑文件的内容。
- -w 关闭自动换行，通过这个参数可以编辑内容很长的列。
- -x 关闭换面下方的命令列表。
- -z 让pico可被Ctrl+z中断，暂存在后台作业里。
- +<列数编号> 执行pico指令进入编辑模式时，从指定的列数开始编辑。

实例

使用pico命令来编辑testfile文件，在终端中输入如下命令：

```
pico testfile
```

输出结果如下：

```
GNU nano 2.0.9 文件: testfile #从左到右分别为编辑器版本号、文件名
#编辑区
Linux networks are becoming more and more common, but security is often an over$
Linux Network Securty focuses on securing Linux in a networked environment, whe$
[ 已读取3 行] #以下为菜单栏
^G 求助^O 写入^R 读档^Y 上页^K 剪切文字^C 在标位置
^X 离开^J 对齐^W 搜寻^V 下页^U 还原剪切^T 拼写检查
```

Linux rgrep命令

Linux rgrep命令用于递归查找文件里符合条件的字符串。

rgrep指令的功能和grep指令类似，可查找内容包含指定的范本样式的文件，如果发现某文件的内容符合所指定的范本样式，预设rgrep指令会把含有范本样式的那一列显示出来。

语法

```
rgrep [-?BcDFhHilnNrv][-R<范本样式>][-W<列长度>][-x<扩展名>][--help][--version][范本样式][文件
```

参说明数：

- -? 显示范本样式与范例的说明。
- -B 忽略二进制的数据。
- -c 计算符合范本样式的列数。
- -D 排错模式，只列出指令搜寻的目录清单，而不会读取文件内容。
- -F 当遇到符号连接时，rgrep预设是忽略不予处理，加上本参数后，rgrep指令就会读取该连接所指向的原始文件的内容。
- -h 特别将符合范本样式的字符串标示出来。
- -H 只列出符合范本样式的字符串，而非显示整列的内容。
- -i 忽略字符大小写的差别。
- -l 列出文件内容符合指定的范本样式的文件名称。
- -n 在显示符合坊本样式的那一列之前，标示出该列的列数编号。
- -N 不要递归处理。
- -r 递归处理，将指定目录下的所有文件及子目录一并处理。
- -R<范本样式> 此参数的效果和指定"-r"参数类似，但只主力符合范本样式文件名称的文件。
- -v 反转查找。
- -W<列长度> 限制符合范本样式的字符串所在列，必须拥有的字符数。
- -x<扩展名> 只处理符合指定扩展名的文件名称的文件。
- --help 在线帮助。
- --version 显示版本信息。

实例

在当前目录下查找句子中包含"Hello"字符串的文件，可使用如下命令：

```
rgrep Hello *
```

其搜索结果如下：

```
$ rgrep Hello *             #在当前目录下查找句子中包含“Hello”字符串的文件
testfile_1:Hello 95         #testfile_1中包含“Hello”字符串的句子
testfile_2:Hello 2005       #testfile_2中包含“Hello”字符串的句子
```

Linux sed命令

Linux sed命令是利用script来处理文本文件。

sed可依照script的指令，来处理、编辑文本文件。

语法

```
sed [-hnV][-e<script>][-f<script文件>][文本文件]
```

参数说明：

- -e<script>或--expression=<script> 以选项中指定的script来处理输入的文本文件。
- -f<script文件>或--file=<script文件> 以选项中指定的script文件来处理输入的文本文件。
- -h或--help 显示帮助。
- -n或--quiet或--silent 仅显示script处理后的结果。
- -V或--version 显示版本信息。

实例

在testfile文件的第四行后添加一行，并将结果输出到标准输出，在命令行提示符下输入如下命令：

```
sed -e 4a\newLine testfile
```

首先查看testfile中的内容如下：

```
$ cat testfile #查看testfile 中的内容
HELLO LINUX!
Linux is a free unix-type opterating system.
This is a linux testfile!
Linux test
```

使用sed命令后，输出结果如下：

```
$ sed -e 4a\newline testfile #使用sed 在第四行后添加新字符串
HELLO LINUX! #testfile文件原有的内容
Linux is a free unix-type opterating system.
This is a linux testfile!
Linux test
newline
```

Linux sort命令

Linux sort命令用于将文本文件内容加以排序。

sort可针对文本文件的内容，以行为单位来排序。

语法

```
sort [-bcdfimMnr][-o<输出文件>][-t<分隔字符>][+<起始栏位>-<结束栏位>][--help][--verison][文件]
```

参数说明：

- -b 忽略每行前面开始出的空格字符。
- -c 检查文件是否已经按照顺序排序。
- -d 排序时，处理英文字母、数字及空格字符外，忽略其他的字符。
- -f 排序时，将小写字母视为大写字母。
- -i 排序时，除了040至176之间的ASCII字符外，忽略其他的字符。
- -m 将几个排序好的文件进行合并。
- -M 将前面3个字母依照月份的缩写进行排序。
- -n 依照数值的大小排序。
- -o<输出文件> 将排序后的结果存入指定的文件。
- -r 以相反的顺序来排序。
- -t<分隔字符> 指定排序时所用的栏位分隔字符。
- +<起始栏位>-<结束栏位> 以指定的栏位来排序，范围由起始栏位到结束栏位的前一栏位。
- --help 显示帮助。
- --version 显示版本信息。

实例

在使用sort命令以默认的式对文件的行进行排序，使用的命令如下：

```
sort testfile
```

sort 命令将以默认的方式将文本文件的第一列以ASCII 码的次序排列，并将结果输出到标准输出。

使用 cat命令显示testfile文件可知其原有的排序如下：

```
$ cat testfile      #testfile文件原有排序
test 30
Hello 95
Linux 85
```

使用sort命令重排后的结果如下：

```
$ sort testfile #重排结果
Hello 95
Linux 85
test 30
```

Linux spell命令

Linux spell命令可建立拼写检查程序。

spell可从标准输入设备读取字符串，结束后显示拼错的词汇。

语法

```
spell
```

实例

检查文件testfile是否有拼写错误，在命令行提示符下输入如下命令：

```
spell testfile
```

如果文件中有单词拼写错误，则输出如下信息：

```
$ spell testfile   #检查testfile 拼写错误
scurity            #以下为有错误的单词
tp
LANs
Securty
practicl
applcations
necesary
```

如果所检查的文件没有单词拼写错误，那么，命令运行后不会给出任何信息。

检查从标准输入读取的字符串。例如在命令行中输入如下命令：

```
spell
```

按回车键后，输入一串字符串，然后按Ctrl+D 组合键退出spell，屏幕上将显示拼写有错误的单词。如下所示：

```
$ spell #检查标准输入的字符串的拼写错误
hell,this is a linx sustem! #拼写错误的字符串
linx #以下为有拼写错误的单词
sustem
```

Linux tr命令

Linux tr 命令用于转换或删除文件中的字符。

tr 指令从标准输入设备读取数据，经过字符串转译后，将结果输出到标准输出设备。

语法

```
tr [-cdst][--help][--version][第一字符集][第二字符集]
tr [OPTION]…SET1[SET2]
```

参数说明：

- -c, --complement：反选设定字符。也就是符合 SET1 的部份不做处理，不符合的剩余部份才进行转换
- -d, --delete：删除指令字符
- -s, --squeeze-repeats：缩减连续重复的字符成指定的单个字符
- -t, --truncate-set1：削减 SET1 指定范围，使之与 SET2 设定长度相等
- --help：显示程序用法信息
- --version：显示程序本身的版本信息

字符集合的范围：

- \NNN 八进制值的字符 NNN (1 to 3 为八进制值的字符)
- \ 反斜杠
- \a Ctrl-G 铃声
- \b Ctrl-H 退格符
- \f Ctrl-L 走行换页
- \n Ctrl-J 新行
- \r Ctrl-M 回车
- \t Ctrl-I tab键
- \v Ctrl-X 水平制表符
- CHAR1-CHAR2 ：字符范围从 CHAR1 到 CHAR2 的指定，范围的指定以 ASCII 码的次序为基础，只能由小到大，不能由大到小。
- [CHAR*] ：这是 SET2 专用的设定，功能是重复指定的字符到与 SET1 相同长度为止
- [CHAR*REPEAT] ：这也是 SET2 专用的设定，功能是重复指定的字符到设定的 REPEAT 次数为止(REPEAT 的数字采 8 进位制计算，以 0 为开始)
- [:alnum:] ：所有字母字符与数字
- [:alpha:] ：所有字母字符
- [:blank:] ：所有水平空格

- [:cntrl:] ：所有控制字符
- [:digit:] ：所有数字
- [:graph:] ：所有可打印的字符(不包含空格符)
- [:lower:] ：所有小写字母
- [:print:] ：所有可打印的字符(包含空格符)
- [:punct:] ：所有标点字符
- [:space:] ：所有水平与垂直空格符
- [:upper:] ：所有大写字母
- [:xdigit:] ：所有 16 进位制的数字
- [=CHAR=] ：所有符合指定的字符(等号里的 CHAR，代表你可自订的字符)

实例

将文件testfile中的小写字母全部转换成大写字母，此时，可使用如下命令：

```
cat testfile |tr a-z A-Z
```

testfile文件中的内容如下：

```
$ cat testfile         #testfile原来的内容
Linux networks are becoming more and more common,
but scurity is often an overlooked
issue. Unfortunately, in today’s environment all networks
are potential hacker targets,
fro0m tp-secret military research networks to small home LANs.
Linux Network Securty focuses on securing Linux in a
networked environment, where the
security of the entire network needs to be considered
rather than just isolated machines.
It uses a mix of theory and practicl techniques to
teach administrators how to install and
use security applications, as well as how the
applcations work and why they are necesary.
```

使用 tr 命令大小写转换后，得到如下输出结果：

```
$ cat testfile | tr a-z A-Z #转换后的输出
LINUX NETWORKS ARE BECOMING MORE AND MORE COMMON, BUT SCURITY IS OFTEN AN OVERLOOKED
ISSUE. UNFORTUNATELY, IN TODAY’S ENVIRONMENT ALL NETWORKS ARE POTENTIAL HACKER TARGETS,
FROM TP-SECRET MILITARY RESEARCH NETWORKS TO SMALL HOME LANS.
LINUX NETWORK SECURTY FOCUSES ON SECURING LINUX IN A NETWORKED ENVIRONMENT, WHERE THE
SECURITY OF THE ENTIRE NETWORK NEEDS TO BE CONSIDERED RATHER THAN JUST ISOLATED MACHINES.
IT USES A MIX OF THEORY AND PRACTICL TECHNIQUES TO TEACH ADMINISTRATORS HOW TO INSTALL AN
USE SECURITY APPLICATIONS, AS WELL AS HOW THE APPLCATIONS WORK AND WHY THEY ARE NECESARY.
```

大小写转换，也可以通过[:lower][:upper]参数来实现。例如使用如下命令：

```
cat testfile |tr [:lower:] [:upper:]
```

输出结果如下：

```
$ cat testfile | tr [:lower:] [:upper:] #转换后的输出
LINUX NETWORKS ARE BECOMING MORE AND MORE COMMON, BUT SCURITY IS OFTEN AN OVERLOOKED
ISSUE. UNFORTUNATELY, IN TODAY’S ENVIRONMENT ALL NETWORKS ARE POTENTIAL HACKER TARGETS,
FROM TP-SECRET MILITARY RESEARCH NETWORKS TO SMALL HOME LANS.
LINUX NETWORK SECURTY FOCUSES ON SECURING LINUX IN A NETWORKED ENVIRONMENT, WHERE THE
SECURITY OF THE ENTIRE NETWORK NEEDS TO BE CONSIDERED RATHER THAN JUST ISOLATED MACHINES.
IT USES A MIX OF THEORY AND PRACTICL TECHNIQUES TO TEACH ADMINISTRATORS HOW TO INSTALL AN
USE SECURITY APPLICATIONS, AS WELL AS HOW THE APPLCATIONS WORK AND WHY THEY ARE NECESARY.
```

Linux expr命令

expr命令是一个手工命令行计数器，用于在UNIX/LINUX下求表达式变量的值，一般用于整数值，也可用于字符串。

语法

```
expr 表达式
```

表达式说明:

- 用空格隔开每个项；
- 用 / (反斜杠) 放在 shell 特定的字符前面；
- 对包含空格和其他特殊字符的字符串要用引号括起来

实例

1、计算字串长度

```
> expr length “this is a test”
 14
```

2、抓取字串

```
> expr substr “this is a test” 3 5
is is
```

3、抓取第一个字符数字串出现的位置

```
> expr index "sarasara"  a
 2
```

4、整数运算

```
> expr 14 % 9
5
> expr 10 + 10
20
> expr 1000 + 900
1900
> expr 30 / 3 / 2
5
> expr 30 /* 3 (使用乘号时，必须用反斜线屏蔽其特定含义。因为shell可能会误解显示星号的意义)
90
> expr 30 * 3
expr: Syntax error
```

Linux uniq命令

Linux uniq命令用于检查及删除文本文件中重复出现的行列。

uniq可检查文本文件中重复出现的行列。

语法

```
uniq [-cdu][-f<栏位>][-s<字符位置>][-w<字符位置>][--help][--version][输入文件][输出文件]
```

参数：

- -c或--count 在每列旁边显示该行重复出现的次数。
- -d或--repeated 仅显示重复出现的行列。
- -f<栏位>或--skip-fields=<栏位> 忽略比较指定的栏位。
- -s<字符位置>或--skip-chars=<字符位置> 忽略比较指定的字符。
- -u或--unique 仅显示出一次的行列。
- -w<字符位置>或--check-chars=<字符位置> 指定要比较的字符。
- --help 显示帮助。
- --version 显示版本信息。
- [输入文件] 指定已排序好的文本文件。
- [输出文件] 指定输出的文件。

实例

文件testfile中第2 行、第5 行、第9 行为相同的行，使用uniq 命令删除重复的行，可使用以下命令：

```
uniq testfile
```

testfile中的原有内容为：

```
$ cat testfile      #原有内容
test 30
test 30
test 30
Hello 95
Hello 95
Hello 95
Hello 95
Linux 85
Linux 85
```

使用uniq 命令删除重复的行后，有如下输出结果：

```
$ uniq testfile     #删除重复行后的内容
test 30
Hello 95
Linux 85
```

检查文件并删除文件中重复出现的行，并在行首显示该行重复出现的次数。使用如下命令：

```
uniq-c testfile
```

结果输出如下：

```
$ uniq-ctestfile      #删除重复行后的内容
3 test 30             #前面的数字的意义为该行共出现了3次
4 Hello 95            #前面的数字的意义为该行共出现了4次
2 Linux 85            #前面的数字的意义为该行共出现了2次
```

Linux wc命令

Linux wc命令用于计算字数。

利用wc指令我们可以计算文件的Byte数、字数、或是列数，若不指定文件名称、或是所给予的文件名为"-"，则wc指令会从标准输入设备读取数据。

语法

```
wc [-clw][--help][--version][文件...]
```

参数：

- -c或--bytes或--chars 只显示Bytes数。
- -l或--lines 只显示列数。
- -w或--words 只显示字数。
- --help 在线帮助。
- --version 显示版本信息。

实例

在默认的情况下，wc将计算指定文件的行数、字数，以及字节数。使用的命令为：

```
wc testfile
```

先查看testfile文件的内容，可以看到：

```
$ cat testfile
Linux networks are becoming more and more common, but scurity is often an overlooked
issue. Unfortunately, in today’s environment all networks are potential hacker targets,
fro0m tp-secret military research networks to small home LANs.
Linux Network Securty focuses on securing Linux in a networked environment, where the
security of the entire network needs to be considered rather than just isolated machines.
It uses a mix of theory and practicl techniques to teach administrators how to install an
use security applications, as well as how the applcations work and why they are necesary.
```

使用 wc统计，结果如下：

```
$ wc testfile           # testfile文件的统计信息
3 92 598 testfile       # testfile文件的行数为3、单词数92、字节数598
```

其中，3 个数字分别表示testfile文件的行数、单词数，以及该文件的字节数。

如果想同时统计多个文件的信息，例如同时统计testfile、testfile_1、testfile_2，可使用如下命令：

```
wc testfile testfile_1 testfile_2   #统计三个文件的信息
```

输出结果如下：

```
$ wc testfile testfile_1 testfile_2  #统计三个文件的信息
3 92 598 testfile                    #第一个文件行数为3、单词数92、字节数598
9 18 78 testfile_1                   #第二个文件的行数为9、单词数18、字节数78
3 6 32 testfile_2                    #第三个文件的行数为3、单词数6、字节数32
15 116 708 总用量                    #三个文件总共的行数为15、单词数116、字节数708
```

Linux命令大全 - 文件传输

lprm	lpr	lpq	lpd
bye	ftp	uuto	uupick
uucp	uucico	tftp	ncftp
ftpshut	ftpwho	ftpcount	

Linux lprm命令

Linux lprm命令用于将一个工作由打印机贮列中移除

尚未完成的打印机工作会被放在打印机贮列之中，这个命令可用来将常未送到打印机的工作取消。由于每一个打印机都有一个独立的贮列，你可以用 -P 这个命令设定想要作用的印列机。如果没有设定的话，会使用系统预设的打印机。

这个命令会检查使用者是否有足够的权限删除指定的档案，一般而言，只有档案的拥有者或是系统管理员才有这个权限。

语法

```
/usr/bin/lprm [P] [file...]
```

实例

将打印机 hpprinter 中的第 1123 号工作移除

```
lprm -Phpprinter 1123
```

将第 1011 号工作由预设印表机中移除

```
lprm 1011
```

Linux lpr命令

lpr(line printer，按行打印)实用程序用来将一个或多个文件放入打印队列等待打印。

lpr 可以用来将料资送给本地或是远端的主机来处理。

语法

```
lpr [ -P printer ]
```

参数：

- -p Printer: 将资料送至指定的打印机 Printer，预设值为 lp。

实例

下面的命令行将在名为mailroom的打印机上打印report文件：

```
$ lpr -P mailroom report
```

使用一条打印命令可打印多个文件，下面的命令行在名为laser1的打印机上打印3个文件：

```
$ lpr -P laser1 05.txt 108.txt 12.txt
```

Linux lpq命令

Linux lpq命令用于查看一个打印队列的状态，该程序可以查看打印机队列状态及其所包含的打印任务。

语法

lpq [l] [P] [user]

参数说明：

- -P 指定一个打印机，否则使用默认打印机或环境变量PRINTER指定的打印机
- -l 打印组成作业的所有文件的信息。。

实例

为系统默认的打印机printer的一个空队列。

```
$ lpq
printer is ready
no entries
```

如果事先并未指定打印机（使用-P选项），系统便会显示默认的打印机。如果向打印机发送打印任务，然后查看打印队列，便会看到如下列表。

```
$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
[me@linuxbox ~]$ lpq
printer is ready and printing
Rank     Owner    Job     File(s)                         Total Size
active   me       603     (stdin)
```

Linux lpd命令

Linux lpd命令 是一个常驻的打印机管理程序，它会根据 /etc/printcap 的内容来管理本地或远端的打印机。

/etc/printcap 中定义的每一个打印机必须在 /var/lpd 中有一个相对应的目录，目录中以 cf 开头的档案表示一个等待送到适当装置的印表工作。这个档案通常是由 lpr 所产生。

lpr 和 lpd 组成了一个可以离线工作的系统，当你使用 lpr 时，打印机不需要能立即可用，甚至不用存在。

lpd 会自动监视打印机的状况，当打印机上线后，便立即将档案送交处理。这个得所有的应用程序不必等待打印机完成前一工作。

语法

```
lpd [-l] [#port]
```

参数说明：

- -l: 将一些除错讯息显示在标准输出上。

- # **port:** 一般而言，**lpd** 会使用 **getservbyname** 取得适当的 **TCP/IP port**，你可以使用这个参数强迫 **lpd** 使用指定的 **port**。

实例

这个程序通常是由 /etc/rc.d 中的程序在系统启始阶段执行。

Linux bye命令

Linux bye命令用于中断FTP连线并结束程序。

在ftp模式下，输入bye即可中断目前的连线作业，并结束ftp的执行。

语法

```
bye
```

Linux ftp命令

Linux ftp命令设置文件系统相关功能。

FTP是ARPANet的标准文件传输协议，该网络就是现今Internet的前身。

语法

```
ftp [-dignv][主机名称或IP地址]
```

参数：

- -d 详细显示指令执行过程，便于排错或分析程序执行的情形。
- -i 关闭互动模式，不询问任何问题。
- -g 关闭本地主机文件名称支持特殊字符的扩充特性。
- -n 不使用自动登陆。
- -v 显示指令执行过程。

实例

例如使用ftp命令匿名登录ftp.kernel.org服务器，该服务是Linux 内核的官方服务器，可以使用如下命令：

```
ftp ftp.kernel.org #发起链接请求
```

Linux ncftp命令

Linux ncftp命令用于传输文件。

FTP让用户得以下载存放于服务器主机的文件，也能将文件上传到远端主机放置。

NcFTP是文字模式FTP程序的佼佼者，它具备多样特色， 包括显示传输速率，下载进度，自动续传，标住书签，可通过防火墙和代理服务器等。

当不指定用户名时，ncftp 命令会自动尝试使用匿名账户anonymous 去连接远程FTP 服 务器，不需要用户输入账号和密码。

语法

```
ncftp [主机或IP地址]
```

参数说明：

- -u<用户名> 指定登录FTP服务器的用户名
- -p<密码> 设置用户密码
- -P<端口号> 指定FTP端口号，默认为21
- -j<账号> 指定账号
- -h 帮助信息
- -v 版本信息

实例

使用ncftp命令匿名连接FTP服务器。

例如想匿名连接ftp.kernel.org服务器，同时不想输入anonymous等匿名用户名，可直接使用ncftp命令：

```
ncftp ftp.kernel.org
```

得到如下信息：

```
$ ncftp ftp.kernel.org #匿名连接ftp.kernel.org服务器
NcFTP 3.2.1 (Jul 29, 2007) by Mike Gleason (http://www.NcFTP.com/contact/).
#ncftp版权、版本等信息
Copyright (c) 1992-2005 by Mike Gleason.
All rights reserved.
Connecting to 149.20.20.133... #连接服务器
Welcome to ftp.kernel.org.
Logging in... #匿名登录
Welcome to the #欢迎信息
LINUX KERNEL ARCHIVES
ftp.kernel.org
"Much more than just kernels"
IF YOU'RE ACCESSING THIS SITE VIA A WEB BROWSER
PLEASE USE THE HTTP URL BELOW INSTEAD!
----> If you are looking for mirror sites, please go <----
----> to mirrors.kernel.org instead <----
This site is provided as a public service by the Linux Kernel
Organization, a California nonprofit corporation. Bandwidth is
provided by The Internet Software Consortium, Inc. Our servers are
located in San Francisco and Palo Alto, California; Corvallis, Oregon;
Amsterdam, Netherlands and Ume., Sweden; use in violation of any
applicable laws strictly prohibited.
Due to U.S. Exports Regulations, all cryptographic software on this
site is subject to the following legal notice:
This site includes publicly available encryption source code
which, together with object code resulting from the compiling of
publicly available source code, may be exported from the United
States under License Exception "TSU" pursuant to 15 C.F.R. Section
740.13(e).
This legal notice applies to cryptographic software only. Please see
the Bureau of Industry and Security (http://www.bis.doc.gov/) for more
information about current U.S. regulations.
Neither the Linux Kernel Organization, nor its sponsors make any
guarantees, explicit or implicit, about the contents of this site.
Use at your own risk.
This site is accessible via the following mechanisms:
FTP ftp://ftp.kernel.org/pub/
HTTP http://www.kernel.org/pub/
RSYNC rsync://rsync.kernel.org/pub/
NFS and SMB/CIFS are no longer available.
For comments on this site, please contact <ftpadmin@kernel.org>.
Please do not use this address for questions that are not related to
the operation of this site. Please see our homepage at
http://www.kernel.org/ for links to Linux documentation resources.
Login successful.
Logged in to ftp.kernel.org.
ncftp / >
```

提示：ncftp的命令提示符为"ncftp / >"，而不是ftp中的"ftp / >"。

使用ncftp命令操作、下载文件。

ncftp的命令基本上与ftp相同，例如可以使用"cd"命令切换在FTP服务器中的当前目录，使用"ls"命令列出当前目录内容，使用"get"命令下载"/pub"目录下的README文件、使用"quit"离开ncftp等。操作结果如下：

```
ncftp / > pwd                #查看当前路径
ftp://ftp.kernel.org         #当前路径为根目录
ncftp / > ls                 #查看当前目录列表
bin/ for_mirrors_only/ pub/
dev/ lib/ usr@
etc/ lost+found/ welcome.msg@
ncftp / > cd pub             #切换目录到pub 子目录
Directory successfully changed.
ncftp /pub > ls              #查看pub 的目录列表
dist/ media/ scm/
index.html RCS/ site/
linux/ README software/
lost+found/ README_ABOUT_BZ2_FILES tools/
ncftp /pub > get README      #下载README 文件
README: 1.87 KB 10.39 KB/s
ncftp /pub > quit            #离开ncftp
```

与ftp不同的是，ncftp此时会提示用户是否将FTP服务器保存为书签，以便于下次登录，用户可以进行自定义书签名等操作，如下所示：

```
You have not saved a bookmark for this site.  #离开提示信息
Would you like to save a bookmark to:
ftp://ftp.kernel.org/pub/
Save? (yes/no) yes                            #确认是否保存
Enter a name for this bookmark, or hit enter for "kernel": kernel #输入书签名
Bookmark "kernel" saved.
```

Linux tftp命令

Linux tftp命令用于传输文件。

FTP让用户得以下载存放于远端主机的文件，也能将文件上传到远端主机放置。tftp是简单的文字模式ftp程序，它所使用的指令和FTP类似。

语法

```
tftp [主机名称或IP地址]
```

操作说明：

- connect：连接到远程tftp服务器
- mode：文件传输模式
- put：上传文件
- get：下载文件
- quit：退出
- verbose：显示详细的处理信息
- tarce：显示包路径
- status：显示当前状态信息
- binary：二进制传输模式
- ascii：ascii 传送模式
- rexmt：设置包传输的超时时间
- timeout：设置重传的超时时间
- help：帮助信息
- ？：帮助信息

实例

连接远程服务器"218.28.188.288"，然后使用put 命令下载其中根目录下的文件"README"，可使用命令如下：

```
tftp 218.28.188.288 #连接远程服务器
```

连接服务器之后可进行相应的操作，具体如下：

```
$ tftp 218.28.188.228                        #连接远程服务器
tftp> ?                                      #使用?，参考帮助
Commands may be abbreviated. Commands are: #帮助命令列表
connect connect to remote tftp
mode set file transfer mode
put send file
get receive file
quit exit tftp
verbose toggle verbose mode
trace toggle packet tracing
status show current status
binary set mode to octet
ascii set mode to netascii
rexmt set per-packet retransmission timeout
timeout set total retransmission timeout
? print help information
tftp>get README                               #远程下载README文件
getting from 218.28.188.288 to /home/cmd
Recived 168236 bytes in 1.5 seconds[112157 bit/s]
tftp>quit                                     #离开tftp
```

Linux uuto命令

Linux uuto命令将文件传送到远端的UUCP主机。

uuto为script文件，它实际上会执行uucp，用来将文件传送到远端UUCP主机，并在完成工作后，以邮件通知远端主机上的用户。

语法

```
uuto [文件][目的]
```

参数：

相关参数请参考 uucp指令。

实例

将文件传送到远程UUCP主机localhost的tmp 目录，在命令提示符中直接输入如下命令：

```
uuto./testfile localhost/tmp #将文件传送到远程UUCP 主机localhost的tmp目录
```

该命令通常没有输出。

Linux uupick命令

Linux uupick命令处理传送进来的文件。

当其他主机通过UUCP将文件传送进来时，可利用uupick指令取出这些文件。

语法

```
uupick [-v][-I<配置文件>][-s<主机>][-x<层级>][--help]
```

参数：

- -I<配置文件>或--config<配置文件> 指定配置文件。
- -s<主机>或--system<主机> 处理由指定主机传送过来的文件。
- -v或--version 显示版本信息。
- --help 显示帮助。

实例

处理由主机localhost传送过来的文件。在命令行直接输入如下命令：

```
uupick-s localhost
```

该命令通常没有输出。

Linux uucp命令

Linux uucp命令用于在Unix系统之间传送文件。

UUCP为Unix系统之间，通过序列线来连线的协议。uucp使用UUCP协议，主要的功能为传送文件。

语法

```
uucp [-cCdfjmrRtvW][-g<等级>][-I<配置文件>][-n<用户>][-x<类型>][--help][...来源][目的]
```

参数说明：

- -c或--nocopy 不用将文件复制到缓冲区。
- -C或--copy 将文件复制到缓冲区。
- -d或--directiories 在传送文件时，自动在[目的]建立必要的目录。
- -f或--nodirectiories 在传送文件时，若需要在[目的]建立目录，则放弃执行该作业。
- -g<等级>或--grade<等级> 指定文件传送作业的优先顺序。
- -I<配置文件>或--config<配置文件> 指定uucp配置文件。
- -j或--jobid 显示作业编号。
- -m或--mail 作业结束后，以电子邮件报告作业是否顺利完成。
- -n<用户>或--notify<用户> 作业结束后，以电子邮件向指定的用户报告作业是否顺利完成。
- -r或--nouucico 不要立即启动uucico服务程序，仅将作业送到队列中，待稍后再执行。
- -R或--recursive 若[来源]为目录，则将整个目录包含子目录复制到[目的]。
- -t或--uuto 将最后一个参数视为"主机名!用户"。
- -v或--version 显示版本信息。
- -W或--noexpand 不要将目前所在的目录加入路径。
- -x<类型>或--debug<类型>启动指定的排错模式。
- --help 显示帮助。
- [源...] 指定源文件或路径。
- [目的] 指定目标文件或路径。

实例

将temp/目录下所有文件传送到远程主机localhost的uucp公共目录下的Public/目录下。在命令行中输入如下命令：

```
uucp-d-R temp localhost ~/Public/
```

该命令通常没有输出

Linux uucico命令

Linux uucico命令UUCP文件传输服务程序。

uucico是用来处理uucp或uux送到队列的文件传输工具。uucico有两种工作模式：主动模式和附属模式。当在主动模式下时，uucico会调用远端主机；在附属模式下时，uucico则接受远端主机的调用。

语法

```
uucico [-cCDefqvwz][-i<类型>][-I<文件>][-p<连接端口号码>][-][-rl][-s<主机>][-S<主机>][-u<用户>
```

参数说明

- -c或--quiet 当不执行任何工作时，不要更改记录文件的内容及更新目前的状态。
- -C或--ifwork 当有工作要执行时，才调用-s或-S参数所指定主机。
- -D或--nodetach 不要与控制终端机离线。
- -e或--loop 在附属模式下执行，并且出现要求登入的提示画面。
- -f或--force 当执行错误时，不等待任何时间即重新调用主机。
- -i<类型>或--stdin<类型> 当使用到标准输入设备时，指定连接端口的类型。
- -I<文件>--config<文件> 指定使用的配置文件。
- -l或--prompt 出现要求登入的提示画面。
- -p<连接端口号码>或-port<连接端口号码> 指定连接端口号码。
- -q或--quiet 不要启动uuxqt服务程序。
- -r0或--slave 以附属模式启动。
- -s<主机>或--system<主机> 调用指定的主机。
- -u<用户>或--login<用户> 指定登入的用户帐号，而不允许输入任意的登入帐号。
- -v或--version 显示版本信息，并且结束程序。
- -w或--wait 在主动模式下，当执行调用动作时，则出现要求登入的提示画面。
- -x<类型>或-X<类型>或outgoing-debug<类型> 启动指定的排错模式。
- -z或--try-next 当执行不成功时，尝试下一个选择而不结束程序。
- --help 显示帮助，并且结束程序。

实例

使用主动模式启动uucico服务。在命令提示符下直接输入如下命令：

```
uucico-r1
```

提示：该命令一般没有输出。

Linux ftpshut命令

Linux ftpshut命令在指定的时间关闭FTP服务器。

本指令提供系统管理者在设置的时间关闭FTP服务器，且能在关闭之前发出警告信息通知用户。关闭时间若设置后为"none"，则会马上关闭服务器。如果采用"+30"的方式来设置表示服务器在30分钟之后关闭。依次类推，假设使用"1130"的格式则代表服务器会在每日的11时30分关闭，时间格式为24小时制。FTP服务器关闭后，在/etc目录下会产生一个名称为shutmsg的文件，把它删除后即可再度启动FTP服务器的功能。

语法

```
ftpshut [-d<分钟>][-l<分钟>][关闭时间]["警告信息"]
```

参数：

- -d<分钟> 切断所有FTP连线时间。
- -l<分钟> 停止接受FTP登入的时间。

实例

在晚上11:00 关闭FTP服务器，并在关闭前5 分钟拒绝新的FTP登录，前3 分钟关闭所有ftp的链接，且给出警告信息，可使用如下命令：

```
ftpshut-d 3 -1 5 1100 "Server will be shutdown at 23:00:00"
```

Linux ftpwho命令

Linux ftpwho命令用于显示目前所有以FTP登入的用户信息。

执行这项指令可得知目前用FTP登入系统的用户有那些人，以及他们正在进行的操作。

语法

```
ftpwho
```

参数说明：

- -v 显示版本信息

实例

查询当前有哪些用户正在登录FTP服务器，可直接使用如下命令：

```
ftpwho
```

该命令有如下输出结果：

```
$ ftpwho              #查询当前正在登录FTP 服务器的用户
standalone FTP daemon[2085]:
3547 wyw [1m20s] 1m25s(idle)
Service class - 1 user #当前有一个用户登录FTP服务器
```

Linux ftpcount命令

Linux ftpcount命令用于显示目前以FTP登入的用户人数。

执行这项指令可得知目前用FTP登入系统的人数以及FTP登入人数的上限。

语法

```
ftpcount
```

参数说明：

- -f<设定文件>：指定设定文件的路径。
- -h，--help ：显示帮助信息。

实例

ftpcount 可以直接查询FTP服务器上用户的人数，可直接使用如下命令：

```
ftpcount              #查询当前FTP用户的人数
```

该命令有如下输出结果：

```
$ ftpcount              #查询当前FTP用户的人数
Master proftpd process 2085:
Service class - 6 user #当前共6个用户登录到服务器
```

Linux命令大全 - 磁盘管理

cd	df	dirs	du
edquota	eject	mcd	mdeltree
mdu	mkdir	mlabel	mmd
mrd	mzip	pwd	quota
mount	mmount	rmdir	rmt
stat	tree	umount	ls
quotacheck	quotaoff	lndir	repquota
quotaon			

Linux cd命令

Linux cd命令用于切换当前工作目录至 dirName(目录参数)。

其中 dirName 表示法可为绝对路径或相对路径。若目录名称省略，则变换至使用者的 home 目录 (也就是刚 login 时所在的目录)。

另外，"~" 也表示为 home 目录 的意思，"." 则是表示目前所在的目录，".." 则表示目前目录位置的上一层目录。

语法

```
cd [dirName]
```

- dirName：要切换的目标目录。

实例

跳到 /usr/bin/ :

```
cd /usr/bin
```

跳到自己的 home 目录 :

```
cd ~
```

跳到目前目录的上上两层 :

```
cd ../..
```

Linux df命令

Linux df命令用于显示目前在Linux系统上的文件系统的磁盘使用情况统计。

语法

```
df [选项]... [FILE]...
```

- 文件-a, --all 包含所有的具有 0 Blocks 的文件系统
- 文件--block-size={SIZE} 使用 {SIZE} 大小的 Blocks
- 文件-h, --human-readable 使用人类可读的格式(预设值是不加这个选项的...)
- 文件-H, --si 很像 -h, 但是用 1000 为单位而不是用 1024
- 文件-i, --inodes 列出 inode 资讯，不列出已使用 block
- 文件-k, --kilobytes 就像是 --block-size=1024
- 文件-l, --local 限制列出的文件结构
- 文件-m, --megabytes 就像 --block-size=1048576
- 文件--no-sync 取得资讯前不 sync (预设值)
- 文件-P, --portability 使用 POSIX 输出格式
- 文件--sync 在取得资讯前 sync
- 文件-t, --type=TYPE 限制列出文件系统的 TYPE
- 文件-T, --print-type 显示文件系统的形式
- 文件-x, --exclude-type=TYPE 限制列出文件系统不要显示 TYPE
- 文件-v (忽略)
- 文件--help 显示这个帮手并且离开
- 文件--version 输出版本资讯并且离开

实例

显示文件系统的磁盘使用情况统计：

```
# df
Filesystem     1K-blocks    Used     Available Use% Mounted on
/dev/sda6       29640780 4320704     23814388  16%     /
udev             1536756       4     1536752    1%     /dev
tmpfs             617620     888     616732     1%     /run
none                5120       0     5120       0%     /run/lock
none             1544044     156     1543888    1%     /run/shm
```

第一列指定文件系统的名称，第二列指定一个特定的文件系统1K-块1K是1024字节为单位的总内存。用和可用列正在使用中，分别指定的内存量。

使用列指定使用的内存的百分比，而最后一栏"安装在"指定的文件系统的挂载点。

df也可以显示磁盘使用的文件系统信息：

```
# df test
Filesystem     1K-blocks    Used      Available Use% Mounted on
/dev/sda6       29640780    4320600   23814492  16%       /
```

用一个-i选项的df命令的输出显示inode信息而非块使用量。

```
df -i
Filesystem      Inodes    IUsed    IFree     IUse% Mounted on
/dev/sda6      1884160    261964   1622196   14%        /
udev           212748     560      212188    1%         /dev
tmpfs          216392     477      215915    1%         /run
none           216392     3        216389    1%         /run/lock
none           216392     8        216384    1%         /run/shm
```

显示所有的信息:

```
# df --total
Filesystem     1K-blocks    Used    Available Use% Mounted on
/dev/sda6       29640780 4320720    23814372  16%     /
udev             1536756       4    1536752   1%      /dev
tmpfs             617620     892    616728    1%      /run
none                5120       0    5120      0%      /run/lock
none             1544044     156    1543888   1%      /run/shm
total           33344320 4321772    27516860  14%
```

我们看到输出的末尾，包含一个额外的行，显示总的每一列。

-h选项，通过它可以产生可读的格式df命令的输出：

```
# df -h
Filesystem      Size  Used   Avail Use% Mounted on
/dev/sda6       29G   4.2G   23G   16%     /
udev            1.5G  4.0K   1.5G   1%     /dev
tmpfs           604M  892K   603M   1%     /run
none            5.0M     0   5.0M   0%     /run/lock
none            1.5G  156K   1.5G   1%     /run/shm
```

我们可以看到输出显示的数字形式的'G'（千兆字节），"M"（兆字节）和"K"（千字节）。

这使输出容易阅读和理解，从而使显示可读的。请注意，第二列的名称也发生了变化，为了使显示可读的"大小"。

Linux dirs命令

Linux dirs命令用于显示目录记录。

显示目录堆叠中的记录。

语法

```
dirs [+/-n -l]
```

参数：

- +n 显示从左边算起第n笔的目录。
- -n 显示从右边算起第n笔的目录。
- -l 显示目录完整的记录。

实例

列出"/home/cc/Ruijie"里所有内容的详细信息。可用如下命令。

```
dir -l /home/cc/Ruijie
```

下面是显示的内容：

```
$ dir -l /home/cc/Ruijie

总计2168

-rwxr-xr-x 1 cc cc  112876 2008-06-26 libpcap.so.0.6.2 -rwxr-xr-x 1 cc cc  737192 2008-06

-rwxr-xr-x 1 cc cc 1350772 2005-08-31 xrgsu
```

Linux du命令

Linux du命令用于显示目录或文件的大小。

du会显示指定的目录或文件所占用的磁盘空间。

语法

```
du [-abcDhHklmsSx][-L <符号连接>][-X <文件>][--block-size][--exclude=<目录或文件>][--max-dept
```

参数说明：

- -a或-all 显示目录中个别文件的大小。
- -b或-bytes 显示目录或文件大小时，以byte为单位。
- -c或--total 除了显示个别目录或文件的大小外，同时也显示所有目录或文件的总和。
- -D或--dereference-args 显示指定符号连接的源文件大小。
- -h或--human-readable 以K，M，G为单位，提高信息的可读性。
- -H或--si 与-h参数相同，但是K，M，G是以1000为换算单位。
- -k或--kilobytes 以1024 bytes为单位。
- -l或--count-links 重复计算硬件连接的文件。
- -L<符号连接>或--dereference<符号连接> 显示选项中所指定符号连接的源文件大小。
- -m或--megabytes 以1MB为单位。
- -s或--summarize 仅显示总计。
- -S或--separate-dirs 显示个别目录的大小时，并不含其子目录的大小。
- -x或--one-file-xystem 以一开始处理时的文件系统为准，若遇上其它不同的文件系统目录则略过。
- -X<文件>或--exclude-from=<文件> 在<文件>指定目录或文件。
- --exclude=<目录或文件> 略过指定的目录或文件。
- --max-depth=<目录层数> 超过指定层数的目录后，予以忽略。
- --help 显示帮助。
- --version 显示版本信息。

实例

显示目录或者文件所占空间:

```
# du
608     ./test6
308     ./test4
4       ./scf/lib
4       ./scf/service/deploy/product
4       ./scf/service/deploy/info
12      ./scf/service/deploy
16      ./scf/service
4       ./scf/doc
4       ./scf/bin
32      ./scf
8       ./test3
1288    .
```

只显示当前目录下面的子目录的目录大小和当前目录的总的大小，最下面的1288为当前目录的总大小

显示指定文件所占空间

```
# du log2012.log
300     log2012.log
```

方便阅读的格式显示test目录所占空间情况：

```
# du -h test
608K    test/test6
308K    test/test4
4.0K    test/scf/lib
4.0K    test/scf/service/deploy/product
4.0K    test/scf/service/deploy/info
12K     test/scf/service/deploy
16K     test/scf/service
4.0K    test/scf/doc
4.0K    test/scf/bin
32K     test/scf
8.0K    test/test3
1.3M    test
```

Linux edquota命令

Linux edquota命令用于编辑用户或群组的磁盘配额。

edquota预设会使用vi来编辑使用者或群组的磁盘配额设置。

语法

```
edquota [-p <源用户名称>][-ug][用户或群组名称...]
```

或

```
edquota [-ug] -t
```

参数：

- -u 设置用户的磁盘配额，这是预设的参数。
- -g 设置群组的磁盘配额。
- -p<源用户名称> 将源用户的磁盘配额设置套用至其他用户或群组。
- -t 设置宽限期限。

Linux mlabel命令

Linux mlabel命令用于设定磁盘的标签 (Label)。

如果磁盘上设定过标签，mlabel 会将他显示给使用者。如果没有指定新标签并且没有指定 c 或 s 选项，mlabel 会提示使用者输入新的标签。如果直接按下 Enter ，就会将原本的标签删除。

语法

```
mlabel [-vcs] drive:[new_label]
```

参数说明：

- -v 更多的讯息。
- -c 清除原有的标签，不出现提示讯息。
- -s 显示目前的标签，不出现提示讯息。

实例

将 A 盘的标签更改为 newlabel。

```
mlabel a:newlabel
```

Linux mkdir命令

Linux mkdir命令用于建立名称为 dirName 之子目录。

语法

```
mkdir [-p] dirName
```

参数说明：

- -p 确保目录名称存在，不存在的就建一个。

实例

在工作目录下，建立一个名为 AAA 的子目录：

```
mkdir AAA
```

在工作目录下的 BBB 目录中，建立一个名为 Test 的子目录。 若 BBB 目录原本不存在，则建立一个。（注：本例若不加 -p，且原本 BBB目录不存在，则产生错误。）

```
mkdir -p BBB/Test
```

Linux mdu命令

Linux mdu命令用于显示MS-DOS目录所占用的磁盘空间。

mdu为mstools工具指令，可显示MS-DOS文件系统中目录所占用的磁盘空间。

语法

```
mdu [-as][目录]
```

参数说明：

- -a 显示每个文件及整个目录所占用的空间。
- -s 仅显示整个目录所占用的空间。

Linux mdeltree命令

Linux mdeltree命令可用来删除 MSDOS 格式档案及目录。

mdeltree 会将所指定的目录与目录之下的所有档案与目录都删除掉。如果所指定的档案或目录不存在，则会传回错误讯息。

语法

```
mdeltree [-v] msdosdirectory [msdosdirectories...]
```

参数说明：

- -v 显示更多的信息。

实例

将 A 磁盘根目录中的 msdosdir 目录以下的档案与目录都删除掉。

```
mcopy a:msdosdir
```

Linux mcd命令

Linux mcd为mtools工具指令，可在MS-DOS文件系统中切换工作目录。若不加任何参数，则显示目前所在的磁盘与工作目录。

语法

```
mcd [msdosdirectory]
```

实例

变更目前工作目录到 a: emp 中。

```
mcd a: emp
```

传回目前工作目录。

```
mcd
```

Linux eject命令

Linux eject命令用于退出抽取式设备。

若设备已挂入，则eject会先将该设备卸除再退出。

语法

```
eject [-dfhnqrstv][-a <开关>][-c <光驱编号>][设备]
```

参数说明：

- [设备] 设备可以是驱动程序名称，也可以是挂入点。
- -a<开关>或--auto<开关> 控制设备的自动退出功能。
- -c<光驱编号>或--changerslut<光驱编号> 选择光驱柜中的光驱。
- -d或--default 显示预设的设备，而不是实际执行动作。
- -f或--floppy 退出抽取式磁盘。
- -h或--help 显示帮助。
- -n或--noop 显示指定的设备。
- -q或--tape 退出磁带。
- -r或--cdrom 退出光盘。
- -s或--scsi 以SCSI指令来退出设备。
- -t或--trayclose 关闭光盘的托盘。
- -v或--verbose 执行时，显示详细的说明。

实例

```
# eject //不加参数默认弹出
# eject -r /dev/cdrom //指定设备
```

Linux mount命令

Linux mount命令是经常会使用到的命令，它用于挂载Linux系统外的文件。

语法

```
mount [-hV]
mount -a [-fFnrsvw] [-t vfstype]
mount [-fnrsvw] [-o options [,...]] device | dir
mount [-fnrsvw] [-t vfstype] [-o options] device dir
```

参数说明：

- -V：显示程序版本
- -h：显示辅助讯息
- -v：显示较讯息，通常和 -f 用来除错。
- -a：将 /etc/fstab 中定义的所有档案系统挂上。
- -F：这个命令通常和 -a 一起使用，它会为每一个 mount 的动作产生一个行程负责执行。在系统需要挂上大量 NFS 档案系统时可以加快挂上的动作。
- -f：通常用在除错的用途。它会使 mount 并不执行实际挂上的动作，而是模拟整个挂上的过程。通常会和 -v 一起使用。
- -n：一般而言，mount 在挂上后会在 /etc/mtab 中写入一笔资料。但在系统中没有可写入档案系统存在的情况下可以用这个选项取消这个动作。
- -s-r：等于 -o ro
- -w：等于 -o rw
- -L：将含有特定标签的硬盘分割挂上。
- -U：将档案分割序号为 的档案系统挂下。-L 和 -U 必须在/proc/partition 这种档案存在时才有意义。
- -t：指定档案系统的型态，通常不必指定。mount 会自动选择正确的型态。
- -o async：打开非同步模式，所有的档案读写动作都会用非同步模式执行。
- -o sync：在同步模式下执行。
- -o atime、-o noatime：当 atime 打开时，系统会在每次读取档案时更新档案的『上一次调用时间』。当我们使用 flash 档案系统时可能会选项把这个选项关闭以减少写入的次数。
- -o auto、-o noauto：打开/关闭自动挂上模式。
- -o defaults:使用预设的选项 rw, suid, dev, exec, auto, nouser, and async.
- -o dev、-o nodev-o exec、-o noexec允许执行档被执行。
- -o suid、-o nosuid：
- 允许执行档在 root 权限下执行。
- -o user、-o nouser：使用者可以执行 mount/umount 的动作。

- -o remount：将一个已经挂下的档案系统重新用不同的方式挂上。例如原先是唯读的系统，现在用可读写的模式重新挂上。
- -o ro：用唯读模式挂上。
- -o rw：用可读写模式挂上。
- -o loop=：使用 loop 模式用来将一个档案当成硬盘分割挂上系统。

实例

将 /dev/hda1 挂在 /mnt 之下。

```
#mount /dev/hda1 /mnt
```

将 /dev/hda1 用唯读模式挂在 /mnt 之下。

```
#mount -o ro /dev/hda1 /mnt
```

将 /tmp/image.iso 这个光碟的 image 档使用 loop 模式挂在 /mnt/cdrom之下。用这种方法可以将一般网络上可以找到的 Linux 光 碟 ISO 档在不烧录成光碟的情况下检视其内容。

```
#mount -o loop /tmp/image.iso /mnt/cdrom
```

Linux mmd命令

Linux mmd命令用于在MS-DOS文件系统中建立目录。

mmd为mtools工具指令，模拟MS-DOS的md指令，可在MS-DOS的文件系统中建立目录。

语法

```
mmd [目录...]
```

Linux mrd命令

Linux mrd命令用于删除MS-DOS文件系统中的目录。

mrd为mtools工具指令，模拟MS-DOS的rd指令，可删除MS-DOS的目录。

语法

```
mrd [目录...]
```

Linux mzip命令

Linux mzip命令是Zip/Jaz磁盘驱动器控制指令。

mzip为mtools工具指令，可设置Zip或Jaz磁盘驱动区的保护模式以及执行退出磁盘的动作。

语法

```
mzip [-efpqruwx]
```

参数：

- -e 退出磁盘。
- -f 与-e参数一并使用，不管是否已经挂入磁盘中的文件系统，一律强制退出磁盘。
- -p 设置磁盘的写入密码。
- -q 显示目前的状态。
- -r 将磁盘设为防写状态。
- -u 退出磁盘以前，暂时解除磁盘的保护状态。
- -w 将磁盘设为可写入状态。
- -x 设置磁盘的密码。

Linux pwd命令

Linux pwd命令用于显示工作目录。

执行pwd指令可立刻得知您目前所在的工作目录的绝对路径名称。

语法

```
pwd [--help][--version]
```

参数说明:

- --help 在线帮助。
- --version 显示版本信息。

实例

查看当前所在目录：

```
# pwd
/root/test          #输出结果
```

Linux quota命令

Linux quota命令用于显示磁盘已使用的空间与限制。

执行quota指令，可查询磁盘空间的限制，并得知已使用多少空间。

语法

```
quota [-quvV][用户名称...] 或 quota [-gqvV][群组名称...]
```

参数说明：

- -g 列出群组的磁盘空间限制。
- -q 简明列表，只列出超过限制的部分。
- -u 列出用户的磁盘空间限制。
- -v 显示该用户或群组，在所有挂入系统的存储设备的空间限制。
- -V 显示版本信息。

实例

```
# quota -guvs    <==显示目前执行者（就是 root ）的 quota 值
# quota -uvs test <==显示 test 这个使用者的 quota 值
```

Linux mmount命令

Linux mmount命令用于挂入MS-DOS文件系统。

mmount为mtools工具指令，可根据[mount参数]中的设置，将磁盘内容挂入到Linux目录中。

语法

```
mmount [驱动器代号][mount参数]
```

参数：

- [mount参数]的用法请参考 mount指令。

Linux rmdir命令

Linux rmdir命令删除空的目录。

语法

```
rmdir [-p] dirName
```

参数：

- -p 是当子目录被删除后使它也成为空目录的话，则顺便一并删除。

实例

将工作目录下，名为 AAA 的子目录删除：

```
rmdir AAA
```

在工作目录下的 BBB 目录中，删除名为 Test 的子目录。若 Test 删除后，BBB 目录成为空目录，则 BBB 亦予删除。

```
rmdir -p BBB/Test
```

Linux rmt命令

Linux rmt命令通过进程间通信远程控制磁带机。

通过rmt指令，用户可通过IPC连线，远端操控磁带机的倾倒和还原操作。

语法

```
rmt
```

Linux stat命令

Linux stat命令用于显示inode内容。

stat以文字的格式来显示inode的内容。

语法

```
stat [文件或目录]
```

实例

查看 testfile 文件的inode内容内容，可以用以下命令：

stat testfile

执行以上命令输出结果：

```
# stat testfile                #输入命令
  File: `testfile'
  Size: 102             Blocks: 8          IO Block: 4096   regular file
Device: 807h/2055d      Inode: 1265161     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2014-08-13 14:07:20.000000000 +0800
Modify: 2014-08-13 14:07:07.000000000 +0800
Change: 2014-08-13 14:07:07.000000000 +0800
```

Linux tree命令

Linux tree命令用于以树状图列出目录的内容。

执行tree指令，它会列出指定目录下的所有文件，包括子目录里的文件。

语法

```
tree [-aACdDfFgilnNpqstux][-I <范本样式>][-P <范本样式>][目录...]
```

参数说明：

- -a 显示所有文件和目录。
- -A 使用ASNI绘图字符显示树状图而非以ASCII字符组合。
- -C 在文件和目录清单加上色彩，便于区分各种类型。
- -d 显示目录名称而非内容。
- -D 列出文件或目录的更改时间。
- -f 在每个文件或目录之前，显示完整的相对路径名称。
- -F 在执行文件，目录，Socket，符号连接，管道名称名称，各自加上"*","/","=","@","|"号。
- -g 列出文件或目录的所属群组名称，没有对应的名称时，则显示群组识别码。
- -i 不以阶梯状列出文件或目录名称。
- -I<范本样式> 不显示符合范本样式的文件或目录名称。
- -l 如遇到性质为符号连接的目录，直接列出该连接所指向的原始目录。
- -n 不在文件和目录清单加上色彩。
- -N 直接列出文件和目录名称，包括控制字符。
- -p 列出权限标示。
- -P<范本样式> 只显示符合范本样式的文件或目录名称。
- -q 用"?"号取代控制字符，列出文件和目录名称。
- -s 列出文件或目录大小。
- -t 用文件和目录的更改时间排序。
- -u 列出文件或目录的拥有者名称，没有对应的名称时，则显示用户识别码。
- -x 将范围局限在现行的文件系统中，若指定目录下的某些子目录，其存放于另一个文件系统上，则将该子目录予以排除在寻找范围外。

实例

以树状图列出当前目录结构。可直接使用如下命令：

```
tree
```

该命令有如下输出结果：

```
# tree                   #以树状图列出当前目录结构
.                        #当前目录结构
|-- README
|-- examples.desktop
|-- file
|-- file.new
|-- index.htm
|-- test
| |-- README
| |-- file
| |-- testfile
| |-- testfile1
| |-- xaa
| |-- xab
| |-- xac
| |-- xad
| |-- xae
| |-- xaf
| |-- xag
| |-- xah
| `-- xai
|-- test.tar.gz
|-- test.zip
|-- testfile
|-- testfile.new
|-- testfile.patch
|-- testfile1
|-- testfile2
|-- testfile3
|-- xaa
|-- xab
|-- xac
|-- xad
|-- xae
|-- xaf
|-- xag
|-- xah
|-- xai
|-- \345\205\254\345\205\261\347\232\204
|-- \345\233\276\347\211\207
| |-- 075b5c2bb1628c1a5343c10a.jpg
| |-- 0c978fe989ac787e799757095719d3c4.jpg
| |-- 20050726194826866443.jpg
| |-- 20061113171548785122.jpg
| |-- 2007102221576687.jpg
| |-- 39.jpg
| |-- 434887ec4340916a78f0559a.jpg
| |-- 498da016ac02fb2bc93d6d08.jpg
| |-- 7b284f5a0f854da2f3bf90b204149a34.jpg
| |-- 9196c030d342a68d5edf0e98.jpg
| |-- a56c5a90de15c8a9a977a4cc.jpg
| |-- c74f62167c9d2b244a90a79e.jpg
| `-- img13.jpg
|-- \346\226\207\346\241\243
|-- \346\241\214\351\235\242
|-- \346\250\241\346\235\277
|-- \350\247\206\351\242\221
`-- \351\237\263\344\271\220
8 directories, 48 files           #统计信息，该目录共8个子目录，48个文件
```

Linux umount命令

Linux umount命令用于卸除文件系统。

umount可卸除目前挂在Linux目录中的文件系统。

语法

```
umount [-ahnrvV][-t <文件系统类型>][文件系统]
```

参数：

- -a 卸除/etc/mtab中记录的所有文件系统。
- -h 显示帮助。
- -n 卸除时不要将信息存入/etc/mtab文件中。
- -r 若无法成功卸除，则尝试以只读的方式重新挂入文件系统。
- -t<文件系统类型> 仅卸除选项中所指定的文件系统。
- -v 执行时显示详细的信息。
- -V 显示版本信息。
- [文件系统] 除了直接指定文件系统外，也可以用设备名称或挂入点来表示文件系统。

实例

下面两条命令分别通过设备名和挂载点卸载文件系统，同时输出详细信息：

```
# umount -v /dev/sda1          通过设备名卸载
/dev/sda1 umounted
# umount -v /mnt/mymount/      通过挂载点卸载
/tmp/diskboot.img umounted
```

如果设备正忙，卸载即告失败。卸载失败的常见原因是，某个打开的shell当前目录为挂载点里的某个目录：

```
# umount -v /mnt/mymount/
umount: /mnt/mymount: device is busy
umount: /mnt/mymount: device is busy
```

Linux ls命令

Linux ls命令用于显示指定工作目录下之内容（列出目前工作目录所含之文件及子目录)。

语法

```
ls [-alrtAFR] [name...]
```

参数：

- -a 显示所有文件及目录 (ls内定将文件名或目录名称开头为"."的视为隐藏档，不会列出)
- -l 除文件名称外，亦将文件型态、权限、拥有者、文件大小等资讯详细列出
- -r 将文件以相反次序显示(原定依英文字母次序)
- -t 将文件依建立时间之先后次序列出
- -A 同 -a，但不列出 "." (目前目录) 及 ".." (父目录)
- -F 在列出的文件名称后加一符号；例如可执行档则加 "*", 目录则加 "/"
- -R 若目录下有文件，则以下之文件亦皆依序列出

实例

列出根目录()下的所有目录：

```
# ls /
bin               dev   lib         media  net   root     srv  upload  www
boot              etc   lib64       misc   opt   sbin     sys  usr
home  lost+found  mnt    proc  selinux  tmp  var
```

列出目前工作目录下所有名称是 s 开头的文件，越新的排越后面：

```
ls -ltr s*
```

将 /bin 目录以下所有目录及文件详细资料列出：

```
ls -lR /bin
```

列出目前工作目录下所有文件及目录；目录于名称后加 "/", 可执行档于名称后加 "*"：

```
ls -AF
```

Linux quotacheck命令

Linux quotacheck命令用于检查磁盘的使用空间与限制。

执行quotacheck指令，扫描挂入系统的分区，并在各分区的文件系统根目录下产生quota.user和quota.group文件，设置用户和群组的磁盘空间限制。

语法

```
quotacheck [-adgRuv][文件系统...]
```

参数：

- -a 扫描在/etc/fstab文件里，有加入quota设置的分区。
- -d 详细显示指令执行过程，便于排错或了解程序执行的情形。
- -g 扫描磁盘空间时，计算每个群组识别码所占用的目录和文件数目。
- -R 排除根目录所在的分区。
- -u 扫描磁盘空间时，计算每个用户识别码所占用的目录和文件数目。
- -v 显示指令执行过程。

Linux quotaoff命令

Linux quotaoff命令关闭磁盘空间限制。

执行quotaoff指令可关闭用户和群组的磁盘空间限制。

语法

```
quotaoff [-aguv][文件系统...]
```

参数说明：

- -a 关闭在/etc/fstab文件里，有加入quota设置的分区的空间限制。
- -g 关闭群组的磁盘空间限制。
- -u 关闭用户的磁盘空间限制。
- -v 显示指令执行过程。

实例

关闭配额限制:

```
# quotaoff -a
```

Linux lndir命令

Linux lndir命令用于连接目录内容。

执行lndir指令，可一口气把源目录底下的文件和子目录统统建立起相互对应的符号连接。

语法

```
lndir [-ignorelinks][-silent][源目录][目的目录]
```

参数：

- -ignorelinks 直接建立符号连接的符号连接。
- -silent 不显示指令执行过程。

实例

给目录下所有的文件或者子文件目录建立链接：

```
lndir /home/uptech abc
```

Linux repquota命令

Linux repquota命令用于检查磁盘空间限制的状态。

执行repquota指令，可报告磁盘空间限制的状况，清楚得知每位用户或每个群组已使用多少空间。

语法

```
repquota [-aguv][文件系统...]
```

参数说明：

- -a 列出在/etc/fstab文件里，有加入quota设置的分区的使用状况，包括用户和群组。
- -g 列出所有群组的磁盘空间限制。
- -u 列出所有用户的磁盘空间限制。
- -v 显示该用户或群组的所有空间限制。

Linux quotaon命令

Linux quotaon命令用于开启磁盘空间限制。

执行quotaon指令可开启用户和群组的才磅秒年空间限制，各分区的文件系统根目录必须有quota.user和quota.group配置文件。

语法

```
quotaon [-aguv][文件系统...]
```

参数说明：

- -a 开启在/ect/fstab文件里，有加入quota设置的分区的空间限制。
- -g 开启群组的磁盘空间限制。
- -u 开启用户的磁盘空间限制。
- -v 显示指令指令执行过程。

Linux命令大全 - 磁盘维护

badblocks	cfdisk	dd	e2fsck
ext2ed	fsck	fsck.minix	fsconf
fdformat	hdparm	mformat	mkbootdisk
mkdosfs	mke2fs	mkfs.ext2	mkfs.msdos
mkinitrd	mkisofs	mkswap	mpartition
swapon	symlinks	sync	mbadblocks
mkfs.minix	fsck.ext2	fdisk	losetup
mkfs	sfdisk	swapoff	

Linux badblocks命令

Linux badblocks命令用于检查磁盘装置中损坏的区块。

执行指令时须指定所要检查的磁盘装置，及此装置的磁盘区块数。

语法

```
badblocks [-svw][-b <区块大小>][-o <输出文件>][磁盘装置][磁盘区块数][启始区块]
```

参数说明：

- -b<区块大小> 指定磁盘的区块大小，单位为字节。
- -o<输出文件> 将检查的结果写入指定的输出文件。
- -s 在检查时显示进度。
- -v 执行时显示详细的信息。
- -w 在检查时，执行写入测试。
- [磁盘装置] 指定要检查的磁盘装置。
- [磁盘区块数] 指定磁盘装置的区块总数。
- [启始区块] 指定要从哪个区块开始检查。

实例

查看系统当前硬盘信息。

```
# fdisk -l
```

例如，显示信息如下：

```
Disk /dev/sda: 298.9 GB, 298999349248 bytes
255 heads, 63 sectors/track, 36351 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1               1         262     2104483+  82  Linux swap / Solaris
/dev/sda2   *         263       32898   262148670   83  Linux
/dev/sda3           32899       36351    27736222+  83  Linux

Disk /dev/sdb: 42.9 GB, 42949672960 bytes
64 heads, 32 sectors/track, 40960 cylinders
Units = cylinders of 2048 * 512 = 1048576 bytes
```

通过命令扫描硬盘。

```
# badblocks -s -v /dev/sdnx
```

其中n表示硬盘设备名，x表示硬盘对应的分区号。例如需要检查"/dev/sda2"，执行命令如下：

```
# badblocks -s -v /dev/sda2

Checking blocks 0 to 30681000
Checking for bad blocks (read-only test): 306809600674112/ 306810000000
30680964
30680965
30680966
30680967
30680968
30680969
30680970
30680971
30680972
30680973
...
done
Pass completed, 37 bad blocks found.其中，“37 bad blocks found”表示硬盘存在37个坏块。
```

Linux cfdisk命令

Linux cfdisk命令用于磁盘分区。

cfdisk是用来磁盘分区的程序，它十分类似DOS的fdisk，具有互动式操作界面而非传统fdisk的问答式界面，您可以轻易地利用方向键来操控分区操作。

语法

```
cfdisk [-avz][-c <柱面数目>-h <磁头数目>-s <盘区数目>][-P <r,s,t>][外围设备代号]
```

参数说明：

- -a 在程序里不用反白代表选取，而以箭头表示。
- -c<柱面数目> 忽略BIOS的数值，直接指定磁盘的柱面数目。
- -h<磁头数目> 忽略BIOS的数值，直接指定磁盘的磁头数目。
- -P<r,s,t> 显示分区表的内容，附加参数"r"会显示整个分区表的详细资料，附加参数"s"会依照磁区的顺序显示相关信息，附加参数"t"则会以磁头，磁区，柱面的方式来显示资料。
- -s<磁区数目> 忽略BIOS的数值，直接指定磁盘的磁区数目。
- -v 显示版本信息。
- -z 不读取现有的分区，直接当作没有分区的新磁盘使用。

实例

进行磁盘分区：

```
# cfsik
```

进行磁盘分区，使用箭头进行操作，而不使用反白表示：

```
# cfsik -a
```

进行磁盘分区，使用箭头进行操作，而不使用反白表示：

```
# cfsik -s 3
```

VM59220:1 Resource interpreted as Image but transferred with MIME type text/html: "http://googleads.g.doubleclick.net/pagead/adview?ai=C9xWpIRctVK75CYm28gXc8Y…bC1FEwdf3NGxgwxaMy7yS-2Ac7otU34AGyoyh--HWiPSLAaAGIQ&sigh=-A-s3VVItog&vis=1". ads?client=ca-pub-5751451760833794

Linux dd命令

Linux dd命令用于读取、转换并输出数据。

dd可从标准输入或文件中读取数据，根据指定的格式来转换数据，再输出到文件、设备或标准输出。

参数说明:

- if=文件名：输入文件名，缺省为标准输入。即指定源文件。
- of=文件名：输出文件名，缺省为标准输出。即指定目的文件。
- ibs=bytes：一次读入bytes个字节，即指定一个块大小为bytes个字节。
  obs=bytes：一次输出bytes个字节，即指定一个块大小为bytes个字节。
  bs=bytes：同时设置读入/输出的块大小为bytes个字节。
- cbs=bytes：一次转换bytes个字节，即指定转换缓冲区大小。
- skip=blocks：从输入文件开头跳过blocks个块后再开始复制。
- seek=blocks：从输出文件开头跳过blocks个块后再开始复制。
- count=blocks：仅拷贝blocks个块，块大小等于ibs指定的字节数。
- conv=<关键字>，关键字可以有以下11种：
  - conversion：用指定的参数转换文件。
  - ascii：转换ebcdic为ascii
  - ebcdic：转换ascii为ebcdic
  - ibm：转换ascii为alternate ebcdic
  - block：把每一行转换为长度为cbs，不足部分用空格填充
  - unblock：使每一行的长度都为cbs，不足部分用空格填充
  - lcase：把大写字符转换为小写字符
  - ucase：把小写字符转换为大写字符
  - swab：交换输入的每对字节
  - noerror：出错时不停止
  - notrunc：不截短输出文件
  - sync：将每个输入块填充到ibs个字节，不足部分用空（NUL）字符补齐。
- --help：显示帮助信息
- --version：显示版本信息

实例

在Linux 下制作启动盘，可使用如下命令：

```
dd if=boot.img of=/dev/fd0 bs=1440k
```

将testfile文件中的所有英文字母转换为大写，然后转成为testfile_1文件，在命令提示符中使用如下命令：

```
dd if=testfile_2 of=testfile_1 conv=ucase
```

其中testfile_2 的内容为：

```
$ cat testfile_2 #testfile_2的内容
HELLO LINUX!
Linux is a free unix-type opterating system.
This is a linux testfile!
Linux test
```

转换完成后，testfile_1 的内容如下：

```
$ dd if=testfile_2 of=testfile_1 conv=ucase #使用dd 命令，大小写转换记录了0+1 的读入
记录了0+1 的写出
95字节（95 B）已复制，0.000131446 秒，723 KB/s
cmd@hdd-desktop:~$ cat testfile_1 #查看转换后的testfile_1文件内容
HELLO LINUX!
LINUX IS A FREE UNIX-TYPE OPTERATING SYSTEM.
THIS IS A LINUX TESTFILE!
LINUX TEST #testfile_2中的所有字符都变成了大写字母
```

由标准输入设备读入字符串，并将字符串转换成大写后，再输出到标准输出设备，使用的命令为：

```
dd conv=ucase
```

输入以上命令后按回车键，输入字符串，再按回车键，按组合键Ctrl+D 退出，出现以下结果：

```
$ dd conv=ucase
Hello Linux! #输入字符串后按回车键
HELLO LINUX! #按组合键Ctrl+D退出，转换成大写结果
记录了0+1 的读入
记录了0+1 的写出
13字节（13 B）已复制，12.1558 秒，0.0 KB/s
```

href http://www.w3cschool.cc/linux/linux-comm-e2fsck.html linux-comm-e2fsck.html:31 'Attr.nodeValue' is deprecated. Please use 'value' instead. adsbygoogle.js:32

Linux e2fsck命令

Linux e2fsck命令用于检查使用 Linux ext2 档案系统的 partition 是否正常工作。

语法

```
e2fsck [-pacnydfvFV] [-b superblock] [-B blocksize] [-l|-L bad_blocks_file] [-C fd] devic
```

参数说明：

- device ： 预备检查的硬盘 partition，例如：/dev/sda1
- -a : 对 partition 做检查，若有问题便自动修复，等同 -p 的功能
- -b : 设定存放 superblock 的位置
- -B : 设定单位 block 的大小
- -c : 检查该partition 是否有坏轨
- -C file : 将检查的结果存到 file 中以便查看
- -d : 列印 e2fsck 的 debug 结果
- -f : 强制检查
- -F : 在开始检查前，将device 的 buffer cache 清空，避免有错误发生
- -l bad_blocks_file : 将有坏轨的block资料加到 bad_blocks_file 里面
- -L bad_blocks_file : 设定坏轨的block资料存到 bad_blocks_file 里面，若无该档则自动产生
- -n : 将档案系统以[唯读]方式开启
- -p : 对 partition 做检查，若有问题便自动修复
- -v : 详细显示模式
- -V : 显示出目前 e2fsck 的版本
- -y : 预先设定所有检查时的问题均回答[是]

实例

检查 /dev/hda5 是否正常，如果有异常便自动修复，并且设定若有问答，均回答[是] :

```
e2fsck -a -y /dev/hda5
```

注意 :

大部份使用 e2fsck 来检查硬盘 partition 的情况时，通常都是情形特殊，因此最好先将该 partition umount，然后再执行 e2fsck 来做检查，若是要非要检查 / 时，则请进入 singal user mode 再执行。

Linux ext2ed命令

Linux ext2ed命令是ext2文件系统编辑程序。

ext2ed可直接处理硬盘分区上的数据，这指令只有Red Hat Linux才提供。

语法

```
ext2ed
```

一般指令：

- setdevice[设备名称] 指定要处理的设备。
- disablewrite 将ext2ed设为只读的状态。
- enablewrite 将ext2ed设为可读写的状态。
- help[指令] 显示个别指令的帮助。
- next 移至下一个单位，单位会依目前所在的模式而异。
- prev 移至前一个单位，单位会依目前所在的模式而异。
- pgup 移至下一页。
- pgdn 移至上一页。
- set 修改目前的数据，参数会依目前所在的模式而异。
- writedata 在执行此指令之后，才会实际修改分区中的数据。
- ext2进入3种模式的指令
- super 进入main superblock,即Superblock模式。
- group<编号> 进入指定的group，即Group模式。
- cd<目录或文件> 在inode模式下，进入指定的目录或文件，即Inode模式。
- Superblock模式
- gocopy<备份编号> 进入指定的superblock备份。
- setactivecopy 将目前所在的superblock，复制到main superblock。
- Group模式
- blockbitmap 显示目前groupo的区块图。
- inode 进入目前group的第一个inode。
- inodebitmap 显示目前group的inode二进制码。
- Inode模式
- dir 进入目录模式。
- file 进入文件模式。

Linux mkbootdisk命令

Linux mkbootdisk命令用于建立目前系统的启动盘。

mkbootdisk可建立目前系统的启动盘。

语法

```
mkbootdisk [--noprompt][--verbose][--version][--device <设备>][--mkinitrdargs <参数>][kern
```

参数：

- --device<设备> 指定设备。
- --mkinitrdargs<参数> 设置mkinitrd的参数。
- --noprompt 不会提示用户插入磁盘。
- --verbose 执行时显示详细的信息。
- --version 显示版本信息。

Linux fsck命令

Linux fsck命令用于检查与修复 Linux 档案系统，可以同时检查一个或多个 Linux 档案系统。

语法

```
fsck [-sACVRP] [-t fstype] [--] [fsck-options] filesys [...]
```

参数 ：

- filesys ： device 名称(eg./dev/sda1)，mount 点 (eg. / 或 /usr)
- -t : 给定档案系统的型式，若在 /etc/fstab 中已有定义或 kernel 本身已支援的则不需加上此参数
- -s : 依序一个一个地执行 fsck 的指令来检查
- -A : 对/etc/fstab 中所有列出来的 partition 做检查
- -C : 显示完整的检查进度
- -d : 列印 e2fsck 的 debug 结果
- -p : 同时有 -A 条件时，同时有多个 fsck 的检查一起执行
- -R : 同时有 -A 条件时，省略 / 不检查
- -V : 详细显示模式
- -a : 如果检查有错则自动修复
- -r : 如果检查有错则由使用者回答是否修复

实例

检查 msdos 档案系统的 /dev/hda5 是否正常，如果有异常便自动修复 :

```
fsck -t msdos -a /dev/hda5
```

注意 此指令可与 /etc/fstab 相互参考操作来加以了解。

Linux fsck.minix命令

Linux fsck.minix命令用于检查文件系统并尝试修复错误。

当minix文件系统发生错误时，可用fsck.minix指令尝试加以参考。

语法

```
fsck.minix [-aflmrsv][外围设备代号]
```

参数：

- -a 自动修复文件系统，不询问任何问题。
- -f 强制对该文件系统进行完整检查，纵然该文件系统在概略检查下没有问题。
- -l 列出所有文件名称。
- -m 使用类似MINIX操作系统的警告信息。
- -r 采用互动模式，在执行修复时询问问题，让用户得以确认并决定处理方式。
- -s 显示该分区第一个磁区的相关信息。
- -v 显示指令执行过程。

Linux fsconf命令

Linux fsconf命令用于设置文件系统相关功能。

fsconf是Red Hat Linux发行版专门用来调整Linux各项设置的程序。

语法

```
fsconf [--check]
```

参数：

- --chedk 检查特定文件的权限。

Linux fdformat命令

Linux fdformat命令用于对指定的软碟机装置进行低阶格式化。

使用这个指令对软碟格式化的时候，最好指定像是下面的装置：

- /dev/fd0d360 磁碟机 A:，磁片为 360KB 磁碟
- /dev/fd0h1440 磁碟机 A:，磁片为 1.4MB 磁碟
- /dev/fd1h1200 磁碟机 B:，磁片为 1.2MB 磁碟

如果使用像是 /dev/fd0 之类的装置，如果里面的磁碟不是标准容量，格式化可能会失败。在这种情况之下，使用者可以用 setfdprm 指令先行指定必要参数。

语法

```
fdformat [-n] device
```

参数：

- -n 关闭确认功能。这个选项会关闭格式化之后的确认步骤。

实例

```
fdformat -n /dev/fd0h1440
```

将磁碟机 A 的磁片格式化成 1.4MB 的磁片。并且省略确认的步骤。

Linux hdparm命令

Linux hdparm命令用于显示与设定硬盘的参数。

hdparm可检测，显示与设定IDE或SCSI硬盘的参数。

语法

```
hdparm [-CfghiIqtTvyYZ][-a <快取分区>][-A <0或1>][-c <I/O模式>][-d <0或1>][-k <0或1>][-K <0或
```

参数说明：

- -a<快取分区> 设定读取文件时，预先存入块区的分区数，若不加上<快取分区>选项，则显示目前的设定。
- -A<0或1> 启动或关闭读取文件时的快取功能。
- -c<I/O模式> 设定IDE32位I/O模式。
- -C 检测IDE硬盘的电源管理模式。
- -d<0或1> 设定磁盘的DMA模式。
- -f 将内存缓冲区的数据写入硬盘，并清楚缓冲区。
- -g 显示硬盘的磁轨，磁头，磁区等参数。
- -h 显示帮助。
- -i 显示硬盘的硬件规格信息，这些信息是在开机时由硬盘本身所提供。
- -I 直接读取硬盘所提供的硬件规格信息。
- -k<0或1> 重设硬盘时，保留-dmu参数的设定。
- -K<0或1> 重设硬盘时，保留-APSWXZ参数的设定。
- -m<磁区数> 设定硬盘多重分区存取的分区数。
- -n<0或1> 忽略硬盘写入时所发生的错误。
- -p<PIO模式> 设定硬盘的PIO模式。
- -P<磁区数> 设定硬盘内部快取的分区数。
- -q 在执行后续的参数时，不在屏幕上显示任何信息。
- -r<0或1> 设定硬盘的读写模式。
- -S<时间> 设定硬盘进入省电模式前的等待时间。
- -t 评估硬盘的读取效率。
- -T 平谷硬盘快取的读取效率。
- -u<0或1> 在硬盘存取时，允许其他中断要求同时执行。
- -v 显示硬盘的相关设定。
- -W<0或1> 设定硬盘的写入快取。
- -X<传输模式> 设定硬盘的传输模式。

- -y 使IDE硬盘进入省电模式。
- -Y 使IDE硬盘进入睡眠模式。
- -Z 关闭某些Seagate硬盘的自动省电功能。

实例

显示硬盘的相关设置：

```
# hdparm /dev/sda
 /dev/sda:
 IO_support = 0 (default 16-bit)
 readonly = 0 (off)
 readahead = 256 (on)
 geometry = 19929［柱面数］/255［磁头数］/63［扇区数］, sectors = 320173056［总扇区数］, start =
```

显示硬盘的柱面、磁头、扇区数

```
# hdparm -g /dev/sda
 /dev/sda:
 geometry = 19929［柱面数］/255［磁头数］/63［扇区数］, sectors = 320173056［总扇区数］, start =
```

评估硬盘的读取效率

```
 hdparm -t /dev/sda
 /dev/sda:
 Timing buffered disk reads: 166 MB in 3.03 seconds = 54.85 MB/sec
 [root@linuxso.com ~]# hdparm -t /dev/sda
 /dev/sda:
 Timing buffered disk reads: 160 MB in 3.01 seconds = 53.11 MB/sec
 [root@linuxso.com ~]# hdparm -t /dev/sda
 /dev/sda:
 Timing buffered disk reads: 166 MB in 3.00 seconds = 55.31 MB/sec
```

Linux mformat命令

Linux mformat命令用于对MS-DOS文件系统的磁盘进行格式化。

在已经做过低阶格式化的磁片上建立 DOS 档案系统。如果在编程 mtools 的时候把 USE_2M 的参数打开，部分与 2M 格式相关的参数就会发生作用。否则这些参数（像是 S,2,1,M）不会发生作用。

语法

```
mformat [-t cylinders] [-h heads] [-s sectors] [-l volume_label] [-F] [-I fsVer-sion] [-S
```

参数：

- -t 磁柱（synlider）数
- -h 磁头（head）数
- -s 每一磁轨的磁区数
- -l 标签
- -F 将磁碟格式化为 FAT32 格式，不过这个参数还在实验中。
- -I 设定 FAT32 中的版本号。这当然也还在实验中。
- -S 磁区大小代码，计算方式为 sector = 2^(大小代码+7)
- -c 磁丛（cluster）的磁区数。如果所给定的数字会导致磁丛数超过 FAT 表的限制，mformat 会自动放大磁区数。
- -s
- -M 软件磁区大小。这个数字就是系统回报的磁区大小。通常是和实际的大小相同。
- -a 如果加上这个参数，mformat 会产生一组 Atari 系统的序号给这块软碟。
- -X 将软碟格式化成 XDF 格式。使用前必须先用 xdfcopy 指令对软碟作低阶格式化的动作。
- -C 产生一个可以安装 MS-DOS 档案系统的磁碟影像档（disk image）。当然对一个实体磁碟机下这个参数是没有意义的。
- -H 隐藏磁区的数目。这通常适用在格式化硬盘的分割区时，因为通常一个分割区的前面还有分割表。这个参数未经测试，能不用就不用。
- -n 磁碟序号
- -r 根目录的大小，单位是磁区数。这个参数只对 FAT12 和 FAT16 有效。
- -B 使用所指定的档案或是设备的开机磁区做为这片磁片或分割区的开机磁区。当然当中的硬件参数会随之更动。
- -k 尽量保持原有的开机磁区。
- -0 第 0 轨的资料传输率

- -A 第 0 轨以外的资料传输率
- -2 使用 2m 格式
- -1 不使用 2m 格式

实例

用预设值把 a: （就是 /dev/fd0）里的磁碟片格式化。

```
mformat a:
```

Linux mkdosfs命令

Linux mkdosfs命令用于建立DOS文件系统。

device 指你想要建立 DOS 档案系统的装置代号。像是 /dev/hda1 等等。 block_count 则是你希望配置的区块数。如果 block_count 没有指定则系统会自动替你计算符合该装置大小的区块数。

```
mkdosfs [ -c | -l filename ]
    [ -f number_of_FATs ]
    [ -F FAT_size ]
    [ -i volume_id ]
    [ -m message_file ]
    [ -n volume_name ]
    [ -r root_dir_entry ]
    [ -s sector_per_cluster ]
    [ -v ]
    device
    [ block_count ]
```

参数：

- -c 建立档案系统之前先检查是否有坏轨。
- -l 从得定的档案中读取坏轨记录。
- -f 指定档案配置表（FAT , File Allocation Table)的数量。预设值为 2 。目前 Linux 的 FAT 档案系统不支援超过 2 个 FAT 表。通常这个不需要改。
- -F 指定 FAT 表的大小，通常是 12 或是 16 个位元组。12 位元组通常用于磁碟片，16 位元组用于一般硬盘的分割区，也就是所谓的 FAT16 格式。这个值通常系统会自己选定适当的值。在磁碟片上用 FAT16 通常不会发生作用，反之在硬盘上用 FAT12 亦然。
- -i 指定 Volume ID。一般是一个 4 个位元组的数字，像是 2e203a47 。如果不给系统会自己产生。
- -m 当使用者试图用这片磁片或是分割区开机，而上面没有操作系统时，系统会给使用者一段警告讯息。这个参数就是用来变更这个讯息的。你可以先用档案编辑好，然后用这个参数指定，或是用
- -m -
- 这样系统会要求你直接输入这段文字。要特别注意的是，档案里的字串长度不要超过 418 个字，包括展开的跳栏符号（TAB）和换行符号（换行符号在 DOS 底下算两个字元！）
- -n 指定 Volume Name，就是磁碟标签。如同在 DOS 底下的 format 指令一样，给不给都可以。没有预设值。
- -r 指定根目录底下的最大档案数。这里所谓的档案数包括目录。预设值是在软碟上是 112 或是 224 ， 在硬盘上是 512。没事不要改这个数字。
- -s 每一个磁丛（cluster）的磁区数。必须是 2 的次方数。不过除非你知道你在作什么，这个值不要乱给。

- -v 提供额外的讯息

实例

将 A 槽里的磁碟片格式化为 DOS 格式，并将标签设为 Tester

```
mkdosfs -n Tester /dev/fd0
```

Linux mke2fs命令

Linux mke2fs命令用于建立ext2文件系统。

语法

```
mke2fs [-cFMqrSvV][-b <区块大小>][-f <不连续区段大小>][-i <字节>][-N <inode数>][-l <文件>][-L 
```

参数：

- -b<区块大小> 指定区块大小，单位为字节。
- -c 检查是否有损坏的区块。
- -f<不连续区段大小> 指定不连续区段的大小，单位为字节。
- -F 不管指定的设备为何，强制执行mke2fs。
- -i<字节> 指定"字节/inode"的比例。
- -N<inode数> 指定要建立的inode数目。
- -l<文件> 从指定的文件中，读取文件西中损坏区块的信息。
- -L<标签> 设置文件系统的标签名称。
- -m<百分比值> 指定给管理员保留区块的比例，预设为5%。
- -M 记录最后一次挂入的目录。
- -q 执行时不显示任何信息。
- -r 指定要建立的ext2文件系统版本。
- -R=<区块数> 设置磁盘阵列参数。
- -S 仅写入superblock与group descriptors，而不更改inode able inode bitmap以及block bitmap。
- -v 执行时显示详细信息。
- -V 显示版本信息。

Linux mkfs.ext2命令

功能说明：与 mke2fs命令 相同

Linux mkfs.msdos命令

功能说明：与 mkdosfs 命令 相同。

Linux mkinitrd命令

Linux mkinitrd命令用于建立要载入ramdisk的映像文件。

mkinitrd可建立映像文件，以供Linux开机时载入ramdisk。

语法

```
mkinitrd [-fv][--omit-scsi-modules][--version][--preload=<模块名称>][--with=<模块名称>][映像
```

参数：

- -f 若指定的映像问家名称与现有文件重复，则覆盖现有的文件。
- -v执行时显示详细的信息。
- --omit-scsi-modules 不要载入SCSI模块。
- --preload=<模块名称> 指定要载入的模块。
- --with=<模块名称> 指定要载入的模块。
- --version 显示版本信息。

Linux mkisofs命令

Linux mkisofs命令用于建立ISO 9660映像文件。

mkisofs可将指定的目录与文件做成ISO 9660格式的映像文件，以供刻录光盘。

语法

```
mkisofs [-adDfhJlLNrRTvz][-print-size][-quiet][-A <应用程序ID>][-abstract <摘要文件>][-b <开
```

参数：

- -a或--all mkisofs通常不处理备份文件。使用此参数可以把备份文件加到映像文件中。
- -A<应用程序ID>或-appid<应用程序ID> 指定光盘的应用程序ID。
- -abstract<摘要文件> 指定摘要文件的文件名。
- -b<开机映像文件>或-eltorito-boot<开机映像文件> 指定在制作可开机光盘时所需的开机映像文件。
- -biblio<ISBN文件> 指定ISBN文件的文件名，ISBN文件位于光盘根目录下，记录光盘的ISBN。
- -c<开机文件名称> 制作可开机光盘时，mkisofs会将开机映像文件中的全-eltorito-catalog<开机文件名称>全部内容作成一个文件。
- -C<盘区编号，盘区编号> 将许多节区合成一个映像文件时，必须使用此参数。
- -copyright<版权信息文件> 指定版权信息文件的文件名。
- -d或-omit-period 省略文件后的句号。
- -D或-disable-deep-relocation ISO 9660最多只能处理8层的目录，超过8层的部分，RRIP会自动将它们设置成ISO 9660兼容的格式。使用-D参数可关闭此功能。
- -f或-follow-links 忽略符号连接。
- -h 显示帮助。
- -hide<目录或文件名> 使指定的目录或文件在ISO 9660或Rock RidgeExtensions的系统中隐藏。
- -hide-joliet<目录或文件名> 使指定的目录或文件在Joliet系统中隐藏。
- -J或-joliet 使用Joliet格式的目录与文件名称。
- -l或-full-iso9660-filenames 使用ISO 9660 32字符长度的文件名。
- -L或-allow-leading-dots 允许文件名的第一个字符为句号。
- -log-file<记录文件> 在执行过程中若有错误信息，预设会显示在屏幕上。
- -m<目录或文件名>或-exclude<目录或文件名> 指定的目录或文件名将不会房入映像文件中。
- -M<映像文件>或-prev-session<映像文件> 与指定的映像文件合并。

- -N或-omit-version-number 省略ISO 9660文件中的版本信息。
- -o<映像文件>或-output<映像文件> 指定映像文件的名称。
- -p<数据处理人>或-preparer<数据处理人> 记录光盘的数据处理人。
- -print-size 显示预估的文件系统大小。
- -quiet 执行时不显示任何信息。
- -r或-rational-rock 使用Rock Ridge Extensions，并开放全部文件的读取权限。
- -R或-rock 使用Rock Ridge Extensions。
- -sysid<系统ID> 指定光盘的系统ID。
- -T或-translation-table 建立文件名的转换表，适用于不支持Rock Ridge Extensions的系统上。
- -v或-verbose 执行时显示详细的信息。
- -V<光盘ID>或-volid<光盘ID> 指定光盘的卷册集ID。
- -volset-size<光盘总数> 指定卷册集所包含的光盘张数。
- -volset-seqno<卷册序号> 指定光盘片在卷册集中的编号。
- -x<目录> 指定的目录将不会放入映像文件中。
- -z 建立通透性压缩文件的SUSP记录，此记录目前只在Alpha机器上的Linux有效。

Linux mkswap命令

Linux mkswap命令用于设置交换区(swap area)。

mkswap可将磁盘分区或文件设为Linux的交换区。

语法

```
mkswap [-cf][-v0][-v1][设备名称或文件][交换区大小]
```

参数：

- -c 建立交换区前，先检查是否有损坏的区块。
- -f 在SPARC电脑上建立交换区时，要加上此参数。
- -v0 建立旧式交换区，此为预设值。
- -v1 建立新式交换区。
- [交换区大小] 指定交换区的大小，单位为1024字节。

Linux mpartition命令

Linux mpartition命令用于建立或删除MS-DOS的分区。

mpartition为mtools工具指令，可建立或删除磁盘分区。

语法

```
mpartition [-acdfIprv][-b <磁区数>][-h <磁头数>][l <磁区数>][-s <磁区数>][-t <柱面数>][驱动器代
```

参数：

- -a 将分区设置为可开机分区。
- -b<磁区数> 建立分区时，指定要从第几个磁区开始建立分区。
- -c 建立分区。
- -d 将分区设置为无法开机的分区。
- -f 强制地修改分区而不管检查时发生的错误信息。
- -h<磁头数> 建立分区时，指定分区的磁头数。
- -I 删除全部的分区。
- -l<磁区数> 建立分区时，指定分区的容量大小，单位为磁区数。
- -p 当要重新建立分区时，显示命令列。
- -r 删除分区。
- -s<磁区数> 建立分区时，指定每个磁轨的磁区数。
- -t<柱面数> 建立分区时，指定分区的柱面数。
- -v 与-p参数一并使用，若没有同时下达修改分区的命令，则显示目前分区的状态。

Linux swapon命令

Linux swapon命令用于激活Linux系统中交换空间，Linux系统的内存管理必须使用交换区来建立虚拟内存。

语法

```
/sbin/swapon -a [-v]
/sbin/swapon [-v] [-p priority] specialfile ...
/sbin/swapon [-s]
```

参数说明：

- -h 请帮帮我
- -V 显示版本讯息
- -s 显示简短的装置讯息
- -a 自动启动所有SWAP装置
- -p 设定优先权，你可以在0到32767中间选一个数字给他。或是在 /etc/fstab 里面加上 pri=[value] ([value]就是0~32767中间一个数字)，然后你就可以很方便的直接使用 swapon -a 来启动他们，而且有优先权设定。

swapon 是开启swap.

相对的,便有一个关闭swap的指令,swapoff.

Linux symlinks命令

Linux symlinks命令用于维护符号连接的工具程序。

symlinks可检查目录中的符号连接，并显示符号连接类型。以下为symlinks可判断的符号连接类型：

- absolute：符号连接使用了绝对路径。
- dangling：原始文件已经不存在。
- lengthy：符号连接的路径中包含了多余的"../"。
- messy：符号连接的路径中包含了多余的"/"。
- other_fs：原始文件位于其他文件系统中。
- relative：符号连接使用了相对路径。

语法

```
symlinks [-cdrstv][目录]
```

参数：

- -c 将使用绝对路径的符号连接转换为相对路径。
- -d 移除dangling类型的符号连接。
- -r 检查目录下所有子目录中的符号连接。
- -s 检查lengthy类型的符号连接。
- -t 与-c一并使用时，会显示如何将绝对路径的符号连接转换为相对路径，但不会实际转换。
- -v 显示所有类型的符号连接。

Linux sync命令

Linux sync命令用于数据同步,sync命令是在关闭Linux系统时使用的。

Linux 系统中欲写入硬盘的资料有的时候会了效率起见，会写到 filesystem buffer 中，这个 buffer 是一块记忆体空间，如果欲写入硬盘的资料存于此 buffer 中，而系统又突然断电的话，那么资料就会流失了，sync 指令会将存于 buffer 中的资料强制写入硬盘中。

语法

```
sync
```

Linux mbadblocks命令

Linux mbadblocks命令用于检查MS-DOS文件系统的磁盘是否有损坏的磁区。

mbadblocks为mtools工具指令，可用来扫描MS-DOS文件系统的磁盘驱动器，并标示出损坏的磁区。

语法

```
mbadblocks [驱动器代号]
```

Linux mkfs.minix命令

Linux mkfs.minix命令用于建立Minix文件系统。

mkfs.minix可建立Minix文件系统。

语法

```
mkfs.minix [-cv][-i <inode数目>][-l <文件>][-n <文件名长度>][设备名称][区块数]
```

参数：

- -c 检查是否有损坏的区块。
- -i<inode数目> 指定文件系统的inode总数。
- -l<文件> 从指定的文件中，读取文件系统中损坏区块的信息。
- -n<文件名长度> 指定文件名称长度的上限。
- -v 建立第2版的Minix文件系统。

Linux fsck.ext2命令

Linux fsck.ext2命令用于检查文件系统并尝试修复错误。

当ext2文件系统发生错误时，可用fsck.ext2指令尝试加以修复。

语法

```
fsck.ext2 [-acdfFnprsStvVy][-b <分区第一个磁区地址>][-B <区块大小>][-C <反叙述器>][-I <inode缓冲
```

参数：

- -a 自动修复文件系统，不询问任何问题。
- -b<分区第一个磁区地址> 指定分区的第一个磁区的起始地址，也就是Super Block。
- -B<区块大小> 设置该分区每个区块的大小。
- -c 检查指定的文件系统内，是否存在有损坏的区块。
- -C<反叙述器> 指定反叙述器，fsck.ext2指令会把全部的执行过程，都交由其逆向叙述，便于排错或监控程序执行的情形。
- -d 详细显示指令执行过程，便于排错或分析程序执行的情形。
- -f 强制对该文件系统进行完整检查，纵然该文件系统在慨略检查下没有问题。
- -F 检查文件系统之前，先清理该保存设备块区内的数据。
- -I<inode缓冲区块数> 设置欲检查的文件系统，其inode缓冲区的区块数目。
- -l<损坏区块文件> 把文件中所列出的区块，视为损坏区块并将其标示出来，避免应用程序使用该区块。
- -L<损坏区块文件> 此参数的效果和指定"-l"参数类似，但在参考损坏区块文件标示损坏区块之前，会先将原来标示成损坏区块者统统清楚，即全部重新设置，而非仅是加入新的损坏区块标示。
- -n 把欲检查的文件系统设成只读，并关闭互动模式，否决所有询问的问题。
- -p 此参数的效果和指定"-a"参数相同。
- -P<处理inode大小> 设置fsck.ext2指令所能处理的inode大小为多少。
- -r 此参数将忽略不予处理，仅负责解决兼容性的问题。
- -s 检查文件系统时，交换每对字节的内容。
- -S 此参数的效果和指定"-s"参数类似，但不论该文件系统是否已是标准位顺序，一律交换每对字节的内容。
- -t 显示fsck.ext2指令的时序信息。
- -v 详细显示指令执行过程。
- -V 显示版本信息。
- -y 关闭互动模式，且同意所有询问的问题。

Linux fdisk命令

Linux fdisk是一个创建和维护分区表的程序，它兼容DOS类型的分区表、BSD或者SUN类型的磁盘列表。

语法

```
fdisk [必要参数][选择参数]
```

必要参数：

- -l 列出素所有分区表
- -u 与"-l"搭配使用，显示分区数目

选择参数：

- -s<分区编号> 指定分区
- -v 版本信息

菜单操作说明

- m ：显示菜单和帮助信息
- a ：活动分区标记/引导分区
- d ：删除分区
- l ：显示分区类型
- n ：新建分区
- p ：显示分区信息
- q ：退出不保存
- t ：设置分区号
- v ：进行分区检查
- w ：保存修改
- x ：扩展应用，高级功能

实例

显示当前分区情况：

```
# fdisk -l

Disk /dev/sda: 10.7 GB, 10737418240 bytes
255 heads, 63 sectors/track, 1305 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

  Device Boot   Start     End   Blocks  Id System
/dev/sda1  *      1     13   104391  83 Linux
/dev/sda2       14    1305  10377990  8e Linux LVM

Disk /dev/sdb: 5368 MB, 5368709120 bytes
255 heads, 63 sectors/track, 652 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/sdb doesn't contain a valid partition table
```

显示SCSI硬盘的每个分区情况

```
# fdisk -lu

Disk /dev/sda: 10.7 GB, 10737418240 bytes
255 heads, 63 sectors/track, 1305 cylinders, total 20971520 sectors
Units = sectors of 1 * 512 = 512 bytes

  Device Boot   Start     End   Blocks  Id System
/dev/sda1  *     63   208844   104391  83 Linux
/dev/sda2     208845  20964824  10377990  8e Linux LVM

Disk /dev/sdb: 5368 MB, 5368709120 bytes
255 heads, 63 sectors/track, 652 cylinders, total 10485760 sectors
Units = sectors of 1 * 512 = 512 bytes

Disk /dev/sdb doesn't contain a valid partition table
```

Linux losetup命令

Linux losetup命令用于设置循环设备。

循环设备可把文件虚拟成区块设备，籍以模拟整个文件系统，让用户得以将其视为硬盘驱动器，光驱或软驱等设备，并挂入当作目录来使用。

语法

```
losetup [-d][-e <加密方式>][-o <平移数目>][循环设备代号][文件]
```

参数：

- -d 卸除设备。
- -e<加密方式> 启动加密编码。
- -o<平移数目> 设置数据平移的数目。

实例

（1）创建空的磁盘镜像文件，这里创建一个1.44M的软盘

```
$ dd if=/dev/zero of=floppy.img bs=512 count=2880
```

（2）使用 losetup将磁盘镜像文件虚拟成快设备

```
$ losetup /dev/loop1 floppy.img
```

（3）挂载块设备

```
$ mount /dev/loop0 /tmp
```

经过上面的三步之后，我们就可以通过/tmp目录，像访问真实快设备一样来访问磁盘镜像文件floppy.img。

（4） 卸载loop设备

```
$ umount /tmp
$ losetup -d /dev/loop1
```

Linux mkfs命令

使用方式 : mkfs [-V] [-t fstype] [fs-options] filesys [blocks]

Linux mkfs命令用于在特定的分区上建立 linux 文件系统

参数 ：

- device ： 预备检查的硬盘分区，例如：/dev/sda1
- -V : 详细显示模式
- -t : 给定档案系统的型式，Linux 的预设值为 ext2
- -c : 在制做档案系统前，检查该partition 是否有坏轨
- -l bad_blocks_file : 将有坏轨的block资料加到 bad_blocks_file 里面
- block : 给定 block 的大小

实例

在 /dev/hda5 上建一个 msdos 的档案系统，同时检查是否有坏轨存在，并且将过程详细列出来 :

```
mkfs -V -t msdos -c /dev/hda5
```

将sda6分区格式化为ext3格式

```
mfks -t ext3 /dev/sda6
```

注意：这里的文件系统是要指定的，比如 ext3 ；reiserfs ；ext2 ；fat32 ；msdos 等。

Linux getty命令

Linux getty命令用于设置终端机模式，连线速率和管制线路。

getty指令是UNIX之类操作系统启动时所必须的3个步骤之一。

语法

```
getty [-h][-d<组态配置文件>][-r<延迟秒数>][-t<超时秒数>][-w<等待字符串>][终端机编号][连线速率<终端机
```

参数：

- -c<定义配置文件> 指定定义配置文件，预设为/etc/gettydefs。
- -d<组态配置文件> 指定组态配置文件，预设为/etc/conf.getty。
- -h 当传输速率为0时就强制断线。
- -r<延迟秒数> 设置延迟时间。
- -t<超时秒数> 设置等待登入的时间。
- -w<等待字符串> 设置等待回应的字符串。

实例

开启终端：

```
# getty tty7
```

Linux sfdisk命令

Linux sfdisk命令是硬盘分区工具程序。

sfdisk为硬盘分区工具程序，可显示分区的设置信息，并检查分区是否正常。

语法

```
sfdisk [-?Tvx][-d <硬盘>][-g <硬盘>][-l <硬盘>][-s <分区>][-V <硬盘>]
```

参数：

- -?或--help 显示帮助。
- -d<硬盘> 显示硬盘分区的设置。
- -g<硬盘>或--show-geometry<硬盘> 显示硬盘的CHS参数。
- -l<硬盘> 显示后硬盘分区的相关设置。
- -s<分区> 显示分区的大小，单位为区块。
- -T或--list-types 显示所有sfdisk能辨识的文件系统ID。
- -v或--version 显示版本信息。
- -V<硬盘>或--verify<硬盘> 检查硬盘分区是否正常。
- -x或--show-extend 显示扩展分区中的逻辑分区。

实例

显示分区信息：

```
# sfdisk -l

Disk /dev/sda: 1305 cylinders, 255 heads, 63 sectors/track
Units = cylinders of 8225280 bytes, blocks of 1024 bytes, counting from 0

Device Boot Start End #cyls #blocks Id System
/dev/sda1 * 0+ 12 13- 104391 83 Linux
/dev/sda2 13 1304 1292 10377990 8e Linux LVM
/dev/sda3 0 - 0 0 0 Empty
/dev/sda4 0 - 0 0 0 Empty

Disk /dev/sdb: 652 cylinders, 255 heads, 63 sectors/track

sfdisk: ERROR: sector 0 does not have an msdos signature
/dev/sdb: unrecognized partition
No partitions found
```

Linux swapoff命令

Linux swapoff命令用于关闭系统交换区(swap area)。

swapoff实际上为swapon的符号连接，可用来关闭系统的交换区。

语法

```
swapoff [设备]
```

参数：

- -a 将/etc/fstab文件中所有设置为swap的设备关闭
- -h 帮助信息
- -V 版本信息

实例

显示分区信息:

```
# sfdisk -l //显示分区信息

Disk /dev/sda: 1305 cylinders, 255 heads, 63 sectors/track
Units = cylinders of 8225280 bytes, blocks of 1024 bytes, counting from 0

  Device Boot Start   End  #cyls  #blocks  Id System
/dev/sda1  *   0+   12   13-  104391  83 Linux
/dev/sda2     13  1304  1292  10377990  8e Linux LVM
/dev/sda3     0    -    0      0  0 Empty
/dev/sda4     0    -    0      0  0 Empty

Disk /dev/sdb: 652 cylinders, 255 heads, 63 sectors/track

sfdisk: ERROR: sector 0 does not have an msdos signature
/dev/sdb: unrecognized partition
No partitions found
```

关闭交换分区。

```
# swapoff /dev/sda2 // 关闭交换分区
```

Linux命令大全 - 网络通讯

apachectl	arpwatch	dip	getty
mingetty	uux	telnet	uulog
uustat	ppp-off	netconfig	nc
httpd	ifconfig	minicom	mesg
dnsconf	wall	netstat	ping
pppstats	samba	setserial	talk
traceroute	tty	newaliases	uuname
netconf	write	statserial	efax
pppsetup	tcpdump	ytalk	cu
smbd	testparm	smbclient	shapecfg

Linux apachectl命令

Linux apachectl命令可用来控制Apache HTTP服务器的程序。

apachectl是slackware内附Apache HTTP服务器的script文件，可供管理员控制服务器，但在其他Linux的Apache HTTP服务器不一定有这个文件。

语法

```
apachectl [configtest][fullstatus][graceful][help][restart][start][status][stop]
```

参数：

- configtest 检查设置文件中的语法是否正确。
- fullstatus 显示服务器完整的状态信息。
- graceful 重新启动Apache服务器，但不会中断原有的连接。
- help 显示帮助信息。
- restart 重新启动Apache服务器。
- start 启动Apache服务器。
- status 显示服务器摘要的状态信息。
- stop 停止Apache服务器。

Linux arpwatch命令

Linux arpwatch命令用于监听网络上ARP的记录。

ARP(Address Resolution Protocol)是用来解析IP与网络装置硬件地址的协议。

arpwatch可监听区域网络中的ARP数据包并记录，同时将监听到的变化通过E-mail来报告。

语法

```
arpwatch [-d][-f<记录文件>][-i<接口>][-r<记录文件>]
```

参数：

- -d 启动排错模式。
- -f<记录文件> 设置存储ARP记录的文件，预设为/var/arpwatch/arp.dat。
- -i<接口> 指定监听ARP的接口，预设的接口为eth0。
- -r<记录文件> 从指定的文件中读取ARP记录，而不是从网络上监听。
- -n 指定附加的本地网络
- -u 指定用户和用户组
- -e 发送邮件给指定用户，非默认的root用户
- -s 指定用户名作为返回地址，而不是默认的用户root

实例

监听网卡eth0的ARP信息

```
arpwatch -i eth0
```

监听ARP的信息，将相关信息记录到相应的文件

```
# arpwatch -i eth0 -f a.log //将信息记录到a.log中
```

Linux nc命令

Linux nc命令用于设置路由器。

执行本指令可设置路由器的相关参数。

语法

```
nc [-hlnruz][-g<网关...>][-G<指向器数目>][-i<延迟秒数>][-o<输出文件>][-p<通信端口>][-s<来源位址>]
```

参数说明：

- -g<网关> 设置路由器跃程通信网关，最丢哦可设置8个。
- -G<指向器数目> 设置来源路由指向器，其数值为4的倍数。
- -h 在线帮助。
- -i<延迟秒数> 设置时间间隔，以便传送信息及扫描通信端口。
- -l 使用监听模式，管控传入的资料。
- -n 直接使用IP地址，而不通过域名服务器。
- -o<输出文件> 指定文件名称，把往来传输的数据以16进制字码倾倒成该文件保存。
- -p<通信端口> 设置本地主机使用的通信端口。
- -r 乱数指定本地与远端主机的通信端口。
- -s<来源位址> 设置本地主机送出数据包的IP地址。
- -u 使用UDP传输协议。
- -v 显示指令执行过程。
- -w<超时秒数> 设置等待连线的时间。
- -z 使用0输入/输出模式，只在扫描通信端口时使用。

实例

TCP端口扫描

```
# nc -v -z -w2 192.168.0.3 1-100
192.168.0.3: inverse host lookup failed: Unknown host
(UNKNOWN) [192.168.0.3] 80 (http) open
(UNKNOWN) [192.168.0.3] 23 (telnet) open
(UNKNOWN) [192.168.0.3] 22 (ssh) open
```

扫描192.168.0.3 的端口 范围是 1-100

扫描UDP端口

```
# nc -u -z -w2 192.168.0.1 1-1000 //扫描192.168.0.3 的端口 范围是 1-1000
```

扫描指定端口

```
# nc -nvv 192.168.0.1 80 //扫描 80端口
(UNKNOWN) [192.168.0.1] 80 (?) open
y  //用户输入
```

Linux dip命令

Linux dip命令用于IP拨号连接。

dip可控制调制解调器，以拨号IP的方式建立对外的双向连接。

语法

```
dip [-aikltv][-m<MTU数目>][-p<协议>][拨号script文件]
```

参数说明：

- -a 询问用户名称与密码。
- -i 启动拨号服务器功能。
- -k 删除执行中的dip程序。
- -l 指定要删除的连线，必须配合-k参数一起使用。
- -m<MTU数目> 设置最大传输单位，预设值为296。
- -p<协议> 设置通信协议。
- -t 进入dip的指令模式。
- -v 执行时显示详细的信息。

实例

建立拨号连接

```
$ dip -t
```

Linux mingetty命令

Linux mingetty命令是精简版的getty。

mingetty适用于本机上的登入程序。

语法

```
mingetty [--long-hostname][--noclear][tty]
```

参数说明：

- --long-hostname 显示完整的主机名称。
- --noclear 在询问登入的用户名称之前不要清楚屏幕画面。

Linux netconfig命令

Linux netconfig命令用于设置网络环境。

这是Slackware发行版内附程序，它具有互动式的问答界面，让用户轻易完成网络环境的设置。

语法

```
netconfig
```

Linux ppp-off命令

Linux ppp命令用于关闭ppp连线。

这是Slackware发行版内附的程序，让用户切断PPP的网络连线。

语法

```
ppp-off
```

实例

关闭ppp连线

```
# ppp-off
```

Linux uustat命令

Linux uustat命令用于显示UUCP目前的状况。

执行uucp与uux指令后，会先将工作送到队列，再由uucico来执行工作。uustat可显示，删除或启动队列中等待执行的工作。

语法

```
uustat [-aeiKmMNpqQRv][-B<行数>][-c<指令>][-C<指令>][-I<配置文件>][-k<工作>][-o<小时>][-r<工作
```

参数说明：

- -a或-all 显示全部的UUCP工作。
- -B<行数>或--mail-lines<行数> 与-M或-N参数一并使用，用来指定邮件中要包含多少行的信息。
- -c<指令>或--command<指令> 显示与<指令>有关的工作。
- -C<指令>或--not-command<指令> 显示与<指令>无关的工作。
- -e或--executions 仅显示待执行的工作。
- -i或--prompt 针对队列中的每项工作，询问使用是否要删除工作。
- -I<配置文件>或--config<配置文件> 指定配置文件。
- -k<工作>或--kill<工作> 删除指定的工作。
- -m或--status 删除全部的工作。
- -M或-mail 将状态信息邮寄给UUCP管理员。
- -N或--notify 将状态信息分别邮寄给提出该项工作的用户。
- -o<小时>或--older-than<小时> 显示超过指定时数的工作。
- -p或--ps 显示负责UUCP锁定的程序。
- -q或--list 显示每台远端主机上所要执行工作的状态。
- -Q或--no-list 不显示工作。
- -r<工作>或--rejuvenate<工作> 重新启动指定的工作。
- -R或--rejuvenate-all 重新启动全部的工作。
- -s<主机>或--system<主机> 显示与<主机>有关的工作。
- -S<主机>或--not-system<主机> 显示与<主机>无关的工作。
- -v或--version 显示版本信息。
- -u<用户>或--user<用户> 显示与<用户>有关的工作。
- -U<用户>或--not-user<用户> 显示与<用户>无关的工作。
- -W<附注>或--comment<附注> 要放在邮件信息中的附注。
- -y<小时>或--younger-than<小时> 显示低于指定时数的工作。

- -x<层级>或--debug<层级> 指定排错层级。
- --help 显示帮助。

实例

显示所有任务

```
# uustat -a
```

显示等待的任务

```
# uustat -e
```

Linux uulog命令

Linux uulog命令用于显示UUCP记录文件。

uulog可用来显示UUCP记录文件中记录。

语法

```
uulog [-DFISv][-<行数>][-f<主机>][-I<配置文件>][-n<行数>][-s<主机>][-u<用户>][-X<层级>][--help
```

参数说明：

- -D或--debuglog 显示排错记录。
- -f<主机>或--follow<主机> 与-F参数类似，但仅显示与指定主机相关的记录。
- -I<配置文件>或--config<配置文件> 指定程序的配置文件。
- -<行数>,-n<行数>或--lines<行数> 显示记录文件中，从最后算起指定行数的数值。
- -s<主机> 仅显示记录文件中，与指定文件相关的记录。
- -S或--statslog 显示统计记录。
- -u<用户>或--suer<用户> 仅显示记录文件中，与指定用户相关的记录。
- -v或--version 显示版本信息。
- -X<层级>或--debug<层级> 设定排错层级。
- --help 显示帮助。

实例

显示uucp log信息

```
# uulog
```

Linux wall命令

Linux wall命令会将讯息传给每一个 mesg 设定为 yes 的上线使用者。当使用终端机介面做为标准传入时, 讯息结束时需加上 EOF (通常用 Ctrl+D)。

使用权限：所有使用者。

语法

```
wall [ message ]
```

实例

传讯息"hi" 给每一个使用者

```
wall hi
```

广播消息

```
# wall Ilove

Broadcast message from root (pts/4) (Thu May 27 16:41:09 2014):

Ilove
```

Linux uux命令

Linux uux命令用于在远端的UUCP主机上执行指令。

uux可在远端的UUCP主机上执行指令或是执行本机上的指令，但在执行时会使用远端电脑的文件。

语法

```
uux [-bcCIjlnrvz][-a<地址>][-g<等级>][-s<文件>][-x<层级>][--help][指令]
```

参数说明：

- -或-p或--stdin 直接从键盘读取要执行的指令。
- -a<地址>或--requestor<地址> 执行邮件地址，以便寄送状态信息。
- -b或--erturn-stdin 在屏幕上显示状态信息。
- -c或--nocopy 不用将文件复制到缓冲区。
- -C或--copy 将文件复制到缓冲区。
- -g<等级>或--grade<等级> 指定文件传送作业的优先顺序。
- -I或--config file 指定uux配置文件。
- -j或--jobid 显示作业编号。
- -l或--link 将本机上的文件连接到缓冲区。
- -n或--notification=no 无论发生任何状态，都不寄邮件通知用户。
- -r或--nouucico 不要立即启动uucico服务程序，仅将作业送到队列中，然后再执行。
- -s<文件>或--status<文件> 将完成状态保存为指定的文件。
- -v或--version 显示版本信息。
- -x<层级>或--debug<层级> 指定排错层级。
- -z或--notification=error 若发生错误，则以邮件来通知用户。
- --help 显示帮助。

实例

在远程主机 uucp 执行命令

```
# uux hnlinux! date  /// 在远程主机 指定date命令查看系统时间
```

Linux telnet命令

Linux telnet命令用于远端登入。

执行telnet指令开启终端机阶段作业，并登入远端主机。

语法

```
telnet [-8acdEfFKLrx][-b<主机别名>][-e<脱离字符>][-k<域名>][-l<用户名称>][-n<记录文件>][-S<服务
```

参数说明：

- -8 允许使用8位字符资料，包括输入与输出。
- -a 尝试自动登入远端系统。
- -b<主机别名> 使用别名指定远端主机名称。
- -c 不读取用户专属目录里的.telnetrc文件。
- -d 启动排错模式。
- -e<脱离字符> 设置脱离字符。
- -E 滤除脱离字符。
- -f 此参数的效果和指定"-F"参数相同。
- -F 使用Kerberos V5认证时，加上此参数可把本地主机的认证数据上传到远端主机。
- -k<域名> 使用Kerberos认证时，加上此参数让远端主机采用指定的领域名，而非该主机的域名。
- -K 不自动登入远端主机。
- -l<用户名称> 指定要登入远端主机的用户名称。
- -L 允许输出8位字符资料。
- -n<记录文件> 指定文件记录相关信息。
- -r 使用类似rlogin指令的用户界面。
- -S<服务类型> 设置telnet连线所需的IP TOS信息。
- -x 假设主机有支持数据加密的功能，就使用它。
- -X<认证形态> 关闭指定的认证形态。

实例

登录远程主机

```
# telnet 192.168.0.5

//登录IP为 192.168.0.5 的远程主机
```

Linux netstat命令

Linux netstat命令用于显示网络状态。

利用netstat指令可让你得知整个Linux系统的网络情况。

语法

```
netstat [-acCeFghilMnNoprstuvVwx][-A<网络类型>][--ip]
```

参数说明：

- -a或--all 显示所有连线中的Socket。
- -A<网络类型>或--<网络类型> 列出该网络类型连线中的相关地址。
- -c或--continuous 持续列出网络状态。
- -C或--cache 显示路由器配置的快取信息。
- -e或--extend 显示网络其他相关信息。
- -F或--fib 显示FIB。
- -g或--groups 显示多重广播功能群组组员名单。
- -h或--help 在线帮助。
- -i或--interfaces 显示网络界面信息表单。
- -l或--listening 显示监控中的服务器的Socket。
- -M或--masquerade 显示伪装的网络连线。
- -n或--numeric 直接使用IP地址，而不通过域名服务器。
- -N或--netlink或--symbolic 显示网络硬件外围设备的符号连接名称。
- -o或--timers 显示计时器。
- -p或--programs 显示正在使用Socket的程序识别码和程序名称。
- -r或--route 显示Routing Table。
- -s或--statistice 显示网络工作信息统计表。
- -t或--tcp 显示TCP传输协议的连线状况。
- -u或--udp 显示UDP传输协议的连线状况。
- -v或--verbose 显示指令执行过程。
- -V或--version 显示版本信息。
- -w或--raw 显示RAW传输协议的连线状况。
- -x或--unix 此参数的效果和指定"-A unix"参数相同。
- --ip或--inet 此参数的效果和指定"-A inet"参数相同。

实例

显示详细的网络状况

```
# netstat -a
```

显示当前户籍UDP连接状况

```
# netstat -nu
```

显示UDP端口号的使用情况

```
# netstat -apu
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State     PID/Program name
udp    0   0 *:32768             *:*                   -
udp    0   0 *:nfs              *:*                   -
udp    0   0 *:641              *:*                   3006/rpc.statd
udp    0   0 192.168.0.3:netbios-ns    *:*                   3537/nmbd
udp    0   0 *:netbios-ns         *:*                   3537/nmbd
udp    0   0 192.168.0.3:netbios-dgm    *:*                   3537/nmbd
udp    0   0 *:netbios-dgm         *:*                   3537/nmbd
udp    0   0 *:tftp             *:*                   3346/xinetd
udp    0   0 *:999              *:*                   3366/rpc.rquotad
udp    0   0 *:sunrpc            *:*                   2986/portmap
udp    0   0 *:ipp              *:*                   6938/cupsd
udp    0   0 *:1022             *:*                   3392/rpc.mountd
udp    0   0 *:638              *:*                   3006/rpc.statd
```

显示网卡列表

```
# netstat -i
Kernel Interface table
Iface     MTU Met  RX-OK RX-ERR RX-DRP RX-OVR  TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0     1500  0   181864    0    0    0   141278    0    0    0 BMRU
lo     16436  0   3362    0    0    0    3362    0    0    0 LRU
```

显示组播组的关系

```
# netstat -g
IPv6/IPv4 Group Memberships
Interface    RefCnt Group
--------------- ------ ---------------------
lo       1   ALL-SYSTEMS.MCAST.NET
eth0      1   ALL-SYSTEMS.MCAST.NET
lo       1   ff02::1
eth0      1   ff02::1:ff0a:b0c
eth0      1   ff02::1
```

显示网络统计信息

```
# netstat -s
Ip:
  184695 total packets received
  0 forwarded
  0 incoming packets discarded
```

```
    184687 incoming packets delivered
    143917 requests sent out
    32 outgoing packets dropped
    30 dropped because of missing route
Icmp:
    676 ICMP messages received
    5 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 44
        echo requests: 287
        echo replies: 345
    304 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 17
        echo replies: 287
Tcp:
    473 active connections openings
    28 passive connection openings
    4 failed connection attempts
    11 connection resets received
    1 connections established
    178253 segments received
    137936 segments send out
    29 segments retransmited
    0 bad segments received.
    336 resets sent
Udp:
    5714 packets received
    8 packets to unknown port received.
    0 packet receive errors
    5419 packets sent
TcpExt:
    1 resets received for embryonic SYN_RECV sockets
    ArpFilter: 0
    12 TCP sockets finished time wait in fast timer
    572 delayed acks sent
    3 delayed acks further delayed because of locked socket
    13766 packets directly queued to recvmsg prequeue.
    1101482 packets directly received from backlog
    19599861 packets directly received from prequeue
    46860 packets header predicted
    14541 packets header predicted and directly queued to user
    TCPPureAcks: 12259
    TCPHPAcks: 9119
    TCPRenoRecovery: 0
    TCPSackRecovery: 0
    TCPSACKReneging: 0
    TCPFACKReorder: 0
    TCPSACKReorder: 0
    TCPRenoReorder: 0
    TCPTSReorder: 0
    TCPFullUndo: 0
    TCPPartialUndo: 0
    TCPDSACKUndo: 0
    TCPLossUndo: 0
    TCPLoss: 0
    TCPLostRetransmit: 0
    TCPRenoFailures: 0
    TCPSackFailures: 0
    TCPLossFailures: 0
    TCPFastRetrans: 0
    TCPForwardRetrans: 0
    TCPSlowStartRetrans: 0
    TCPTimeouts: 29
    TCPRenoRecoveryFail: 0
    TCPSackRecoveryFail: 0
    TCPSchedulerFailed: 0
    TCPRcvCollapsed: 0
    TCPDSACKOldSent: 0
    TCPDSACKOfoSent: 0
    TCPDSACKRecv: 0
```

```
   TCPDSACKOfoRecv: 0
   TCPAbortOnSyn: 0
   TCPAbortOnData: 1
   TCPAbortOnClose: 0
   TCPAbortOnMemory: 0
   TCPAbortOnTimeout: 3
   TCPAbortOnLinger: 0
   TCPAbortFailed: 3
   TCPMemoryPressures: 0
```

显示监听的套接口

```
# netstat -l
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address        Foreign Address       State
tcp    0   0 *:32769           *:*              LISTEN
tcp    0   0 *:nfs            *:*              LISTEN
tcp    0   0 *:644            *:*              LISTEN
tcp    0   0 *:1002           *:*              LISTEN
tcp    0   0 *:netbios-ssn        *:*              LISTEN
tcp    0   0 *:sunrpc          *:*              LISTEN
tcp    0   0 vm-dev:ipp         *:*              LISTEN
tcp    0   0 *:telnet          *:*              LISTEN
tcp    0   0 *:601            *:*              LISTEN
tcp    0   0 *:microsoft-ds       *:*              LISTEN
tcp    0   0 *:http           *:*              LISTEN
tcp    0   0 *:ssh            *:*              LISTEN
tcp    0   0 *:https           *:*              LISTEN
udp    0   0 *:32768           *:*
udp    0   0 *:nfs            *:*
udp    0   0 *:641            *:*
udp    0   0 192.168.0.3:netbios-ns   *:*
udp    0   0 *:netbios-ns        *:*
udp    0   0 192.168.0.3:netbios-dgm   *:*
udp    0   0 *:netbios-dgm        *:*
udp    0   0 *:tftp           *:*
udp    0   0 *:999            *:*
udp    0   0 *:sunrpc          *:*
udp    0   0 *:ipp            *:*
udp    0   0 *:1022           *:*
udp    0   0 *:638            *:*
Active UNIX domain sockets (only servers)
Proto RefCnt Flags    Type    State     I-Node Path
unix 2   [ ACC ]   STREAM   LISTENING   10621 @/tmp/fam-root-
unix 2   [ ACC ]   STREAM   LISTENING   7096  /var/run/acpid.socket
unix 2   [ ACC ]   STREAM   LISTENING   9792  /tmp/.gdm_socket
unix 2   [ ACC ]   STREAM   LISTENING   9927  /tmp/.X11-unix/X0
unix 2   [ ACC ]   STREAM   LISTENING   10489 /tmp/ssh-lbUnUf4552/agent.4552
unix 2   [ ACC ]   STREAM   LISTENING   10558 /tmp/ksocket-root/kdeinit__0
unix 2   [ ACC ]   STREAM   LISTENING   10560 /tmp/ksocket-root/kdeinit-:0
unix 2   [ ACC ]   STREAM   LISTENING   10570 /tmp/.ICE-unix/dcop4664-1270815442
unix 2   [ ACC ]   STREAM   LISTENING   10843 /tmp/.ICE-unix/4735
unix 2   [ ACC ]   STREAM   LISTENING   10591 /tmp/ksocket-root/klauncherah3arc.slave-soc
unix 2   [ ACC ]   STREAM   LISTENING   7763  /var/run/iiim/.iiimp-unix/9010
unix 2   [ ACC ]   STREAM   LISTENING   11047 /tmp/orbit-root/linc-1291-0-1e92c8082411
unix 2   [ ACC ]   STREAM   LISTENING   11053 /tmp/orbit-root/linc-128e-0-dc070659cbb3
unix 2   [ ACC ]   STREAM   LISTENING   8020  /var/run/dbus/system_bus_socket
unix 2   [ ACC ]   STREAM   LISTENING   58927 /tmp/mcop-root/vm-dev-2c28-4beba75f
unix 2   [ ACC ]   STREAM   LISTENING   7860  /tmp/.font-unix/fs7100
unix 2   [ ACC ]   STREAM   LISTENING   7658  /dev/gpmctl
unix 2   [ ACC ]   STREAM   LISTENING   10498 @/tmp/dbus-s2MLJGO5Ci
```

Linux dnsconf命令

Linux dnsconf命令用于设置DNS服务器组态。

dnsconf实际上为linuxconf的符号连接，提供图形截面的操作方式，供管理员管理DNS服务器。

语法

```
dnsconf [--deldomain<域>][--delsecondary<域>][--newdomain<域>][--set<主机><IP>][--setcname<
```

参数说明：

- --deldomain<域> 删除域。
- --delsecondary<域> 删除次级域。
- --newdomain<域> 新增域。
- --set<主机><IP> 新增主机记录。
- --setcname<CNAME><主机> 设置<CNAME>。
- --setmx<域><主机> 指定域的邮件主机。
- --setns<域><主机> 指定域的DNS服务器。
- --unset<主机> 删除DNS中某台主机的记录。

Linux mesg命令

Linux mesg命令用于设置终端机的写入权限。

将mesg设置y时，其他用户可利用write指令将信息直接显示在您的屏幕上。

语法

```
mesg [ny]
```

参数：

- n 不允许气筒用户将信息直接显示在你的屏幕上。
- y 允许气筒用户将信息直接显示在你的屏幕上。

实例

允许其他用户发信息到当前终端。

root 的终端

```
# mesg y //在这个终端设置允许发送消息
```

其他普通用户的终端：

```
$ write root pts/4
hello
hello
EOF //Ctrl+D 结束输入
```

root 的终端 终端显示

```
#
Message from root@w3cschool.cc (as hnlinux) on pts/5 at 14:48 ...
hello
EOF
```

Linux httpd命令

Linux httpd命令是Apache HTTP服务器程序。

httpd为Apache HTTP服务器程序。直接执行程序可启动服务器的服务。

语法

```
httpd [-hlLStvVX][-c<httpd指令>][-C<httpd指令>][-d<服务器根目录>][-D<设定文件参数>][-f<设定文件>
```

参数说明：

- -c<httpd指令> 在读取配置文件前，先执行选项中的指令。
- -C<httpd指令> 在读取配置文件后，再执行选项中的指令。
- -d<服务器根目录> 指定服务器的根目录。
- -D<设定文件参数> 指定要传入配置文件的参数。
- -f<设定文件> 指定配置文件。
- -h 显示帮助。
- -l 显示服务器编译时所包含的模块。
- -L 显示httpd指令的说明。
- -S 显示配置文件中的设定。
- -t 测试配置文件的语法是否正确。
- -v 显示版本信息。
- -V 显示版本信息以及建立环境。
- -X 以单一程序的方式来启动服务器。

实例

检查配置文件语法错误

```
# httpd -t
httpd: Could not determine the server's fully qualified domain name, using 127.0.0.1 for
Syntax OK
```

启动httpd

```
httpd
httpd: Could not determine the server's fully qualified domain name, using 127.0.0.1 for
```

显示编译模块

```
# httpd -l
Compiled in modules:
 core.c
 prefork.c
 http_core.c
 mod_so.c
```

显示配置文件

```
# httpd -L>1.log|tail -n 20 1.log
Maximum number of children alive at the same time
Allowed in *.conf only outside , or
ServerLimit (prefork.c)
Maximum value of MaxClients for this run of Apache
Allowed in *.conf only outside , or
KeepAliveTimeout (http_core.c)
Keep-Alive timeout duration (sec)
Allowed in *.conf only outside , or
MaxKeepAliveRequests (http_core.c)
Maximum number of Keep-Alive requests per connection, or 0 for infinite
Allowed in *.conf only outside , or
KeepAlive (http_core.c)
Whether persistent connections should be On or Off
Allowed in *.conf only outside , or
LoadModule (mod_so.c)
a module name and the name of a shared object file to load it from
Allowed in *.conf only outside , or
LoadFile (mod_so.c)
shared object file or library to load into the server at runtime
Allowed in *.conf only outside , or
```

Linux ifconfig命令

Linux ifconfig命令用于显示或设置网络设备。

ifconfig可设置网络设备的状态，或是显示目前的设置。

语法

```
ifconfig [网络设备][down up -allmulti -arp -promisc][add<地址>][del<地址>][<hw<网络设备类型><
```

参数说明：

- add<地址> 设置网络设备IPv6的IP地址。
- del<地址> 删除网络设备IPv6的IP地址。
- down 关闭指定的网络设备。
- <hw<网络设备类型><硬件地址> 设置网络设备的类型与硬件地址。
- io_addr<I/O地址> 设置网络设备的I/O地址。
- irq<IRQ地址> 设置网络设备的IRQ。
- media<网络媒介类型> 设置网络设备的媒介类型。
- mem_start<内存地址> 设置网络设备在主内存所占用的起始地址。
- metric<数目> 指定在计算数据包的转送次数时，所要加上的数目。
- mtu<字节> 设置网络设备的MTU。
- netmask<子网掩码> 设置网络设备的子网掩码。
- tunnel<地址> 建立IPv4与IPv6之间的隧道通信地址。
- up 启动指定的网络设备。
- -broadcast<地址> 将要送往指定地址的数据包当成广播数据包来处理。
- -pointopoint<地址> 与指定地址的网络设备建立直接连线，此模式具有保密功能。
- -promisc 关闭或启动指定网络设备的promiscuous模式。
- [IP地址] 指定网络设备的IP地址。
- [网络设备] 指定网络设备的名称。

实例

显示网络设备信息

```
# ifconfig
eth0   Link encap:Ethernet HWaddr 00:50:56:0A:0B:0C
     inet addr:192.168.0.3 Bcast:192.168.0.255 Mask:255.255.255.0
     inet6 addr: fe80::250:56ff:fe0a:b0c/64 Scope:Link
     UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
     RX packets:172220 errors:0 dropped:0 overruns:0 frame:0
     TX packets:132379 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:1000
     RX bytes:87101880 (83.0 MiB) TX bytes:41576123 (39.6 MiB)
     Interrupt:185 Base address:0x2024

lo    Link encap:Local Loopback
     inet addr:127.0.0.1 Mask:255.0.0.0
     inet6 addr: ::1/128 Scope:Host
     UP LOOPBACK RUNNING MTU:16436 Metric:1
     RX packets:2022 errors:0 dropped:0 overruns:0 frame:0
     TX packets:2022 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:0
     RX bytes:2459063 (2.3 MiB) TX bytes:2459063 (2.3 MiB)
```

启动关闭指定网卡

```
# ifconfig eth0 down
# ifconfig eth0 up
```

为网卡配置和删除IPv6地址

```
# ifconfig eth0 add 33ffe:3240:800:1005::2/ 64 //为网卡诶之IPv6地址

# ifconfig eth0 del 33ffe:3240:800:1005::2/ 64 //为网卡删除IPv6地址
```

用ifconfig修改MAC地址

```
# ifconfig eth0 down //关闭网卡
# ifconfig eth0 hw ether 00:AA:BB:CC:DD:EE //修改MAC地址
# ifconfig eth0 up //启动网卡
# ifconfig eth1 hw ether 00:1D:1C:1D:1E //关闭网卡并修改MAC地址
# ifconfig eth1 up //启动网卡
```

配置IP地址

```
# ifconfig eth0 192.168.1.56
//给eth0网卡配置IP地址
# ifconfig eth0 192.168.1.56 netmask 255.255.255.0
// 给eth0网卡配置IP地址,并加上子掩码
# ifconfig eth0 192.168.1.56 netmask 255.255.255.0 broadcast 192.168.1.255
// 给eth0网卡配置IP地址,加上子掩码,加上个广播地址
```

启用和关闭ARP协议

```
# ifconfig eth0 arp  //开启
# ifconfig eth0 -arp  //关闭
```

设置最大传输单元

```
# ifconfig eth0 mtu 1500
//设置能通过的最大数据包大小为 1500 bytes
```

Linux minicom命令

Linux minicom命令用于调制解调器通信程序。

minicom是一个相当受欢迎的PPP拨号连线程序。

语法

```
minicom [-8lmMostz][-a<on或0ff>][-c<on或off>][-C<取文件>][-d<编号>][-p<模拟终端机>][-S<script
```

参数说明：

- -8 不要修改任何8位编码的字符。
- -a<on或0ff> 设置终端机属性。
- -c<on或off> 设置彩色模式。
- -C<取文件> 指定取文件，并在启动时开启取功能。
- -d<编号> 启动或直接拨号。
- -l 不会将所有的字符都转成ASCII码。
- -m 以Alt或Meta键作为指令键。
- -M 与-m参数类似。
- -o 不要初始化调制解调器。
- -p <模拟终端机> 使用模拟终端机。
- -s 开启程序设置画面。
- -S<script文件> 在启动时，执行指定的script文件。
- -t 设置终端机的类型。
- -z 在终端机上显示状态列。
- [配置文件] 指定minicom配置文件。

Linux traceroute命令

Linux traceroute命令用于显示数据包到主机间的路径。

traceroute指令让你追踪网络数据包的路由途径，预设数据包大小是40Bytes，用户可另行设置。

语法

```
traceroute [-dFlnrvx][-f<存活数值>][-g<网关>...][-i<网络界面>][-m<存活数值>][-p<通信端口>][-s<来
```

参数说明：

- -d 使用Socket层级的排错功能。
- -f<存活数值> 设置第一个检测数据包的存活数值TTL的大小。
- -F 设置勿离断位。
- -g<网关> 设置来源路由网关，最多可设置8个。
- -i<网络界面> 使用指定的网络界面送出数据包。
- -I 使用ICMP回应取代UDP资料信息。
- -m<存活数值> 设置检测数据包的最大存活数值TTL的大小。
- -n 直接使用IP地址而非主机名称。
- -p<通信端口> 设置UDP传输协议的通信端口。
- -r 忽略普通的Routing Table，直接将数据包送到远端主机上。
- -s<来源地址> 设置本地主机送出数据包的IP地址。
- -t<服务类型> 设置检测数据包的TOS数值。
- -v 详细显示指令的执行过程。
- -w<超时秒数> 设置等待远端主机回报的时间。
- -x 开启或关闭数据包的正确性检验。

实例

显示到达目的地的数据包路由

```
# traceroute www.google.com
traceroute: Warning: www.google.com has multiple addresses; using 66.249.89.99
traceroute to www.l.google.com (66.249.89.99), 30 hops max, 38 byte packets
1 192.168.0.1 (192.168.0.1) 0.653 ms 0.846 ms 0.200 ms
2 118.250.4.1 (118.250.4.1) 36.610 ms 58.438 ms 55.146 ms
3 222.247.28.177 (222.247.28.177) 54.809 ms 39.879 ms 19.186 ms
4 61.187.255.253 (61.187.255.253) 18.033 ms 49.699 ms 72.147 ms
5 61.137.2.177 (61.137.2.177) 32.912 ms 72.947 ms 41.809 ms
6 202.97.46.5 (202.97.46.5) 60.436 ms 25.527 ms 40.023 ms
7 202.97.35.69 (202.97.35.69) 40.049 ms 66.091 ms 44.358 ms
8 202.97.35.110 (202.97.35.110) 42.140 ms 70.913 ms 41.144 ms
9 202.97.35.14 (202.97.35.14) 116.929 ms 57.081 ms 60.336 ms
10 202.97.60.34 (202.97.60.34) 54.871 ms 69.302 ms 64.353 ms
11 * * *
12 209.85.255.80 (209.85.255.80) 95.954 ms 79.844 ms 76.052 ms
   MPLS Label=385825 CoS=5 TTL=1 S=0
13 209.85.249.195 (209.85.249.195) 118.687 ms 120.905 ms 113.936 ms
14 72.14.236.126 (72.14.236.126) 115.843 ms 137.109 ms 186.491 ms
15 nrt04s01-in-f99.1e100.net (66.249.89.99) 168.024 ms 140.551 ms 161.127 ms
```

Linux talk命令

Linux talk命令用于与其他使用者对谈。

使用权限：所有使用者。

语法

```
talk person [ttyname]
```

参数说明：

- person : 预备对谈的使用者帐号，如果该使用者在其他机器上，则可输入 person@machine.name
- ttyname : 如果使用者同时有两个以上的 tty 连线，可以自行选择合适的 tty 传讯息

实例

与现在机器上的使用者Rollaend对谈，此时 Rollaend 只有一个连线

```
talk Rollaend
```

接下来就是等Rollaend回应，若Rollaend接受，则Rollaend输入 `talk jzlee` 即可开始对谈，结束请按 ctrl+c

与linuxfab.cx上的使用者Rollaend对谈，使用pts/2来对谈

```
talk Rollaend@linuxfab.cx pts/2
```

接下来就是等Rollaend回应，若Rollaend接受，则Rollaend输入 `talk jzlee@jzlee.home` 即可开始对谈，结束请按 ctrl+c

注意：若萤幕的字会出现不正常的字元，试著按 ctrl+l 更新萤幕画面。

Linux ping命令

Linux ping命令用于检测主机。

执行ping指令会使用ICMP传输协议，发出要求回应的信息，若远端主机的网络功能没有问题，就会回应该信息，因而得知该主机运作正常。

语法

```
ping [-dfnqrRv][-c<完成次数>][-i<间隔秒数>][-I<网络界面>][-l<前置载入>][-p<范本样式>][-s<数据包大
```

参数说明：

- -d 使用Socket的SO_DEBUG功能。
- -c<完成次数> 设置完成要求回应的次数。
- -f 极限检测。
- -i<间隔秒数> 指定收发信息的间隔时间。
- -I<网络界面> 使用指定的网络界面送出数据包。
- -l<前置载入> 设置在送出要求信息之前，先行发出的数据包。
- -n 只输出数值。
- -p<范本样式> 设置填满数据包的范本样式。
- -q 不显示指令执行过程，开头和结尾的相关信息除外。
- -r 忽略普通的Routing Table，直接将数据包送到远端主机上。
- -R 记录路由过程。
- -s<数据包大小> 设置数据包的大小。
- -t<存活数值> 设置存活数值TTL的大小。
- -v 详细显示指令的执行过程。

实例

检测是否与主机连通

```
# ping www.w3cschool.cc //ping主机
PING aries.m.alikunlun.com (114.80.174.110) 56(84) bytes of data.
64 bytes from 114.80.174.110: icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from 114.80.174.110: icmp_seq=2 ttl=64 time=0.036 ms
64 bytes from 114.80.174.110: icmp_seq=3 ttl=64 time=0.034 ms
64 bytes from 114.80.174.110: icmp_seq=4 ttl=64 time=0.034 ms
64 bytes from 114.80.174.110: icmp_seq=5 ttl=64 time=0.028 ms
64 bytes from 114.80.174.110: icmp_seq=6 ttl=64 time=0.028 ms
64 bytes from 114.80.174.110: icmp_seq=7 ttl=64 time=0.034 ms
64 bytes from 114.80.174.110: icmp_seq=8 ttl=64 time=0.034 ms
64 bytes from 114.80.174.110: icmp_seq=9 ttl=64 time=0.036 ms
64 bytes from 114.80.174.110: icmp_seq=10 ttl=64 time=0.041 ms

--- aries.m.alikunlun.com ping statistics ---
10 packets transmitted, 30 received, 0% packet loss, time 29246ms
rtt min/avg/max/mdev = 0.021/0.035/0.078/0.011 ms

//需要手动终止Ctrl+C
```

指定接收包的次数

```
# ping -c 2 www.w3cschool.cc
PING aries.m.alikunlun.com (114.80.174.120) 56(84) bytes of data.
64 bytes from 114.80.174.120: icmp_seq=1 ttl=54 time=6.18 ms
64 bytes from 114.80.174.120: icmp_seq=2 ttl=54 time=15.4 ms

--- aries.m.alikunlun.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1016ms
rtt min/avg/max/mdev = 6.185/10.824/15.464/4.640 ms

//收到两次包后，自动退出
```

多参数使用

```
# ping -i 3 -s 1024 -t 255 g.cn //ping主机
PING g.cn (203.208.37.104) 1024(1052) bytes of data.
1032 bytes from bg-in-f104.1e100.net (203.208.37.104): icmp_seq=0 ttl=243 time=62.5 ms
1032 bytes from bg-in-f104.1e100.net (203.208.37.104): icmp_seq=1 ttl=243 time=63.9 ms
1032 bytes from bg-in-f104.1e100.net (203.208.37.104): icmp_seq=2 ttl=243 time=61.9 ms

--- g.cn ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 6001ms
rtt min/avg/max/mdev = 61.959/62.843/63.984/0.894 ms, pipe 2
[root@linux ~]#

//-i 3 发送周期为 3秒 -s 设置发送包的大小 -t 设置TTL值为 255
```

Linux pppstats命令

Linux pppstats命令用于显示PPP连线状态。

利用pppstats(point to point protocol status)指令可让你得知PPP连接网络的相关信息。

语法

```
pppstats [-adrv][-c<执行次数>][-w<间隔秒数>][网络界面]
```

参数说明：

- -a 显示绝对统计值。
- -c<执行次数> 设置回报状况的次数。
- -d 显示相对统计值。
- -r 显示数据包压缩比率的统计值。
- -v 显示VJTCP文件头的压缩效率统计值。
- -w<间隔秒数> 设置显示统计信息的间隔时间。

实例

显示ppp的了连接状态

```
# pppstats
```

Linux samba命令

Linux samba命令用于Samba服务器控制。

samba为script文件，可启动，停止Samba服务器或回报目前的状态。

语法

```
samba [start][stop][status][restart]
```

参数说明：

- start 启动Samba服务器的服务。
- stop 停止Samba服务器的服务。
- status 显示Samba服务器目前的状态。
- restart 重新启动Samba服务器。

实例

启动Samba

```
# samba start
```

Linux statserial命令

Linux statserial命令用于显示串口状态。

statserial(status ofserial port)可显示各个接脚的状态，常用来判断串口是否正常。

语法

```
statserial [-dnx][串口设备名称]
```

参数说明：

- -d 以10进制数字来表示串口的状态。
- -n 仅显示一次串口的状态后即结束程序。
- -x 与-n参数类似，但是以16进制来表示。

实例

显示串口状态

```
# statserial /dev/tty1
```

只显示一次串口状态

```
# statserial -n /dev/tty1
```

Linux write命令

Linux write命令用于传讯息给其他使用者。

使用权限：所有使用者。

语法

```
write user [ttyname]
```

参数说明：

- user : 预备传讯息的使用者帐号
- ttyname : 如果使用者同时有两个以上的 tty 连线，可以自行选择合适的 tty 传讯息

实例

传讯息给 Rollaend，此时 Rollaend 只有一个连线

```
write Rollaend
```

接下来就是将讯息打上去，结束请按 ctrl+c

传讯息给 Rollaend，Rollaend 的连线有 pts/2，pts/3

```
write Rollaend pts/2
```

接下来就是将讯息打上去，结束请按 ctrl+c

注意：若对方设定 mesg n，则此时讯席将无法传给对方。

Linux setserial命令

Linux setserial命令用于设置或显示串口的相关信息。

setserial可用来设置串口或显示目前的设置。

语法

```
setserial [-abgGqvVz][设备][串口参数]
```

参数说明：

- -a 显示详细信息。
- -b 显示摘要信息。
- -g 显示串口的相关信息。
- -G 以指令列表的格式来显示信息。
- -q 执行时显示较少的信息。
- -v 执行时显示较多的信息。
- -V 显示版本信息。
- -z 设置前，先将所有的标记归零。

实例

显示串口信息

```
# setserial -g /dev/ttyS2
/dev/ttyS2, UART: unknown, Port: 0x03e8, IRQ: 4
```

Linux tty命令

Linux tty命令用于显示终端机连接标准输入设备的文件名称。

在Linux操作系统中，所有外围设备都有其名称与代号，这些名称代号以特殊文件的类型存放于/dev目录下。你可以执行tty(teletypewriter)指令查询目前使用的终端机的文件名称。

语法

```
tty [-s][--help][--version]
```

参数说明：

- -s或--silent或--quiet 不显示任何信息，只回传状态代码。
- --help 在线帮助。
- --version 显示版本信息。

实例

显示当前终端

```
# tty
/dev/pts/4
```

Linux newaliases命令

Linux newaliases命令会使用一个在 /etc/aliases 中的档案做使用者名称转换的动作。当 sendmail 收到一个要送给 xxx 的信时，它会依据 aliases档的内容送给另一个使用者。这个功能可以创造一个只有在信件系统内才有效的使用者。例如 mailing list 就会用到这个功能，在 mailinglist 中，我们可能会创造一个叫 redlinux@link.ece.uci.edu 的 mailinglist，但实际上并没有一个叫 redlinux 的使用者。实际 aliases 档的内容是将送给这个使用者的信都收给 mailing list 处理程序负责分送的工作。

/etc/aliases 是一个文字模式的档案，sendmail 需要一个二进位格式的 /etc/aliases.db。newaliases 的功能传是将 /etc/aliases 转换成一个 sendmail 所能了解的数据库。

使用权限：系统管理者。

语法

```
newaliases
```

参数说明：没有任何参数。

实例

```
# newaliases
```

下面命令会做相同的事

```
# sendmail -bi
```

Linux uuname命令

Linux uuname命令用于显示全部的UUCP远端主机。

uuname可显示UUCP远端主机。

语法

```
uuname [-alv][-I<配置文件>][--help]
```

参数说明：

- -a或--aliases 显示别名。
- -I<配置文件>或--config<配置文件> 指定程序的配置文件。
- -l或--local 显示本机名称。
- -v或--version 显示版本信息。
- --help 显示帮助。

实例

显示uucp主机名称

```
# uuname
```

Linux netconf命令

Linux netconf命令用于设置各项网络功能。

netconf是Red Hat Linux发行版专门用来调整Linux各项设置的程序。

语法

```
netconf
```

Linux smbd命令

Linux smbd命令用于Samba服务器程序。

smbd为Samba服务器程序，可分享文件与打印机等网络资源供Windows相关的用户端程序存取。

语法

```
smbd [-aDhoP][-d<排错层级>][-i<范围>][-l<记录文件>][-O<连接槽选项>][-p<连接端口编号>][-s<配置文件
```

参数说明：

- -a 所有的连线记录都会加到记录文件中。
- -d<排错层级> 指定记录文件所记载事件的详细程度。
- -D 使用此参数时，smbd会以服务程序的方式在后台执行。
- -h 显示帮助。
- -i<范围> 指定NetBIOS名称的范围。
- -l<记录文件> 指定记录文件的名称。
- -o 每次启动时，会覆盖原有的记录文件。
- -O<连接槽选项> 设置连接槽选项。
- -p<连接端口编号> 设置连接端口编号。
- -P 仅用来测试smbd程序的正确性。
- -s<配置文件> 指定smbd的设置文件。

实例

启动Samba服务器

```
# smbd -D
```

Linux ytalk命令

Linux ytalk命令用于与其他用户交谈。

通过ytalk指令，你可以和其他用户线上交谈，如果想和其他主机的用户交谈，在用户名称后加上其主机名称或IP地址即可。

语法

```
ytalk [-isxY][-h<主机名称IP地址>][用户名称...]
```

参数说明：

- -h<主机名称IP地址> 指定交谈对象所在的远端主机。
- -i 用提醒声响代替显示信息。
- -s 在指令提示符号先开启ytalk交谈窗。
- -x 关闭图形界面。
- -Y 所有必须回应yes或no的问题，都必须用大写英文字母"Y"或"N"回答。

实例

发送消息

```
# who  //显示当前用户
root   :0      Apr 9 20:17
root   pts/1    Apr 9 20:17
w3c   pts/6    May 27 16:47 (192.168.0.1)
root   pts/2    May 27 17:37 (192.168.0.1)
# ytalk w3c //发送消息
hey
```

Linux tcpdump命令

Linux tcpdump命令用于倾倒网络传输数据。

执行tcpdump指令可列出经过指定网络界面的数据包文件头，在Linux操作系统中，你必须是系统管理员。

语法

```
tcpdump [-adeflnNOpqStvx][-c<数据包数目>][-dd][-ddd][-F<表达文件>][-i<网络界面>][-r<数据包文件>
```

参数说明：

- -a 尝试将网络和广播地址转换成名称。
- -c<数据包数目> 收到指定的数据包数目后，就停止进行倾倒操作。
- -d 把编译过的数据包编码转换成可阅读的格式，并倾倒到标准输出。
- -dd 把编译过的数据包编码转换成C语言的格式，并倾倒到标准输出。
- -ddd 把编译过的数据包编码转换成十进制数字的格式，并倾倒到标准输出。
- -e 在每列倾倒资料上显示连接层级的文件头。
- -f 用数字显示网际网络地址。
- -F<表达文件> 指定内含表达方式的文件。
- -i<网络界面> 使用指定的网络截面送出数据包。
- -l 使用标准输出列的缓冲区。
- -n 不把主机的网络地址转换成名字。
- -N 不列出域名。
- -O 不将数据包编码最佳化。
- -p 不让网络界面进入混杂模式。
- -q 快速输出，仅列出少数的传输协议信息。
- -r<数据包文件> 从指定的文件读取数据包数据。
- -s<数据包大小> 设置每个数据包的大小。
- -S 用绝对而非相对数值列出TCP关联数。
- -t 在每列倾倒资料上不显示时间戳记。
- -tt 在每列倾倒资料上显示未经格式化的时间戳记。
- -T<数据包类型> 强制将表达方式所指定的数据包转译成设置的数据包类型。
- -v 详细显示指令执行过程。
- -vv 更详细显示指令执行过程。
- -x 用十六进制字码列出数据包资料。
- -w<数据包文件> 把数据包数据写入指定的文件。

实例

显示TCP包信息

```
# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
23:35:55.129998 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 148872068:148872168(100) ack 418
23:35:55.182357 IP 192.168.0.1.2101 > 192.168.0.3.ssh: . ack 100 win 64240
23:35:55.182397 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 100:200(100) ack 1 win 2100
23:35:55.131713 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: 50226+ PTR? 1.0.168.192.in-a
23:35:55.131896 PPPoE [ses 0x1cb0] IP 118.250.6.85.64215 > dns2.cs.hn.cn.domain: 50226+ P
23:35:55.154238 PPPoE [ses 0x1cb0] IP dns2.cs.hn.cn.domain > 118.250.6.85.64215: 50226 NX
23:35:55.156298 IP dns2.cs.hn.cn.domain > 192.168.0.3.32804: 50226 NXDomain 0/0/0 (42)
23:35:55.159292 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: 30304+ PTR? 3.0.168.192.in-a
23:35:55.159449 PPPoE [ses 0x1cb0] IP 118.250.6.85.64215 > dns2.cs.hn.cn.domain: 30304+ P
23:35:55.179816 PPPoE [ses 0x1cb0] IP dns2.cs.hn.cn.domain > 118.250.6.85.64215: 30304 NX
23:35:55.181279 IP dns2.cs.hn.cn.domain > 192.168.0.3.32804: 30304 NXDomain 0/0/0 (42)
23:35:55.181806 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 200:268(68) ack 1 win 2100
23:35:55.182177 IP 192.168.0.1.2101 > 192.168.0.3.ssh: . ack 268 win 64198
23:35:55.182677 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: 43983+ PTR? 112.96.103.202.i
23:35:55.182807 PPPoE [ses 0x1cb0] IP 118.250.6.85.64215 > dns2.cs.hn.cn.domain: 43983+ P
23:35:55.183055 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 268:352(84) ack 1 win 2100
23:35:55.201096 PPPoE [ses 0x1cb0] IP dns2.cs.hn.cn.domain > 118.250.6.85.64215: 43983 1/
23:35:55.203087 IP dns2.cs.hn.cn.domain > 192.168.0.3.32804: 43983 1/0/0 (72)
23:35:55.204666 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 352:452(100) ack 1 win 2100
23:35:55.204852 IP 192.168.0.1.2101 > 192.168.0.3.ssh: . ack 452 win 64152
23:35:55.205305 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 452:520(68) ack 1 win 2100
23:35:55.205889 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: 9318+ PTR? 85.6.250.118.in-a
23:35:55.206071 PPPoE [ses 0x1cb0] IP 118.250.6.85.64215 > dns2.cs.hn.cn.domain: 9318+ PT
23:35:55.215338 PPPoE [ses 0x1cb0] IP 115.238.1.45.3724 > 118.250.6.85.64120: P 239275192
23:35:55.216273 IP 115.238.1.45.3724 > 192.168.0.65.2057: P 2392751922:2392751987(65) ack
23:35:55.329204 IP 192.168.0.1.2101 > 192.168.0.3.ssh: . ack 520 win 64135
23:35:55.458214 IP 192.168.0.65.2057 > 115.238.1.45.3724: . ack 65 win 32590
23:35:55.458221 PPPoE [ses 0x1cb0] IP 118.250.6.85.64120 > 115.238.1.45.3724: . ack 65 wi
23:35:55.708228 PPPoE [ses 0x1cb0] IP 115.238.1.45.3724 > 118.250.6.85.64120: P 65:118(53
23:35:55.710213 IP 115.238.1.45.3724 > 192.168.0.65.2057: P 65:118(53) ack 1 win 54
23:35:55.865151 IP 192.168.0.65.2057 > 115.238.1.45.3724: . ack 118 win 32768
23:35:55.865157 PPPoE [ses 0x1cb0] IP 118.250.6.85.64120 > 115.238.1.45.3724: . ack 118 w
23:35:56.242805 IP 192.168.0.65.2057 > 115.238.1.45.3724: P 1:25(24) ack 118 win 32768
23:35:56.242812 PPPoE [ses 0x1cb0] IP 118.250.6.85.64120 > 115.238.1.45.3724: P 1:25(24)
23:35:56.276816 PPPoE [ses 0x1cb0] IP 115.238.1.45.3724 > 118.250.6.85.64120: . ack 25 wi
23:35:56.278240 IP 115.238.1.45.3724 > 192.168.0.65.2057: . ack 25 win 54
23:35:56.349747 PPPoE [ses 0x1cb0] IP 115.238.1.45.3724 > 118.250.6.85.64120: P 118:159(4
23:35:56.351780 IP 115.238.1.45.3724 > 192.168.0.65.2057: P 118:159(41) ack 25 win 54
23:35:56.400051 PPPoE [ses 0x1cb0] IP 119.147.18.44.8000 > 118.250.6.85.4000: UDP, length
23:35:56.475050 IP 192.168.0.65.2057 > 115.238.1.45.3724: . ack 159 win 32762
23:35:56.475063 PPPoE [ses 0x1cb0] IP 118.250.6.85.64120 > 115.238.1.45.3724: . ack 159 w
23:35:56.508968 PPPoE [ses 0x1cb0] IP 115.238.1.45.3724 > 118.250.6.85.64120: P 159:411(2
23:35:56.510182 IP 115.238.1.45.3724 > 192.168.0.65.2057: P 159:411(252) ack 25 win 54
23:35:56.592028 PPPoE [ses 0x1cb0] IP 117.136.2.43.38959 > 118.250.6.85.63283: UDP, lengt

44 packets captured
76 packets received by filter
0 packets dropped by kernel
```

显示指定数量包

```
# tcpdump -c 20
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
23:36:28.949538 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 148875984:148876020(36) ack 4184
23:36:28.994325 IP 192.168.0.1.2101 > 192.168.0.3.ssh: . ack 36 win 64020
23:36:28.994368 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 36:72(36) ack 1 win 2100
23:36:28.950779 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: 18242+ PTR? 1.0.168.192.in-a
23:36:28.950948 PPPoE [ses 0x1cb0] IP 118.250.6.85.64215 > dns2.cs.hn.cn.domain: 18242+ P
23:36:28.960105 PPPoE [ses 0x1cb0] IP 222.82.119.41.13594 > 118.250.6.85.63283: UDP, leng
23:36:28.962192 IP 222.82.119.41.13594 > 192.168.0.65.13965: UDP, length 36
23:36:28.963118 IP 192.168.0.65.13965 > 222.82.119.41.13594: UDP, length 34
23:36:28.963123 PPPoE [ses 0x1cb0] IP 118.250.6.85.63283 > 222.82.119.41.13594: UDP, leng
23:36:28.970185 PPPoE [ses 0x1cb0] IP dns2.cs.hn.cn.domain > 118.250.6.85.64215: 18242 NX
23:36:28.970413 IP dns2.cs.hn.cn.domain > 192.168.0.3.32804: 18242 NXDomain 0/0/0 (42)
23:36:28.972352 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: 17862+ PTR? 3.0.168.192.in-a
23:36:28.972474 PPPoE [ses 0x1cb0] IP 118.250.6.85.64215 > dns2.cs.hn.cn.domain: 17862+ P
23:36:28.982287 PPPoE [ses 0x1cb0] IP 121.12.131.163.13109 > 118.250.6.85.63283: UDP, len
23:36:28.984162 IP 121.12.131.163.13109 > 192.168.0.65.13965: UDP, length 27
23:36:28.985021 IP 192.168.0.65.13965 > 121.12.131.163.13109: UDP, length 103
23:36:28.985027 PPPoE [ses 0x1cb0] IP 118.250.6.85.63283 > 121.12.131.163.13109: UDP, len
23:36:28.991919 PPPoE [ses 0x1cb0] IP dns2.cs.hn.cn.domain > 118.250.6.85.64215: 17862 NX
23:36:28.993142 IP dns2.cs.hn.cn.domain > 192.168.0.3.32804: 17862 NXDomain 0/0/0 (42)
23:36:28.993574 IP 192.168.0.3.ssh > 192.168.0.1.2101: P 72:140(68) ack 1 win 2100
20 packets captured
206 packets received by filter
129 packets dropped by kernel
```

精简显示

```
# tcpdump -c 10 -q //精简模式显示 10个包
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
23:43:05.792280 IP 192.168.0.3.ssh > 192.168.0.1.2101: tcp 36
23:43:05.842115 IP 192.168.0.1.2101 > 192.168.0.3.ssh: tcp 0
23:43:05.845074 IP 115.238.1.45.3724 > 192.168.0.65.2057: tcp 0
23:43:05.907155 IP 192.168.0.3.ssh > 192.168.0.1.2101: tcp 36
23:43:05.793880 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: UDP, length 42
23:43:05.794076 PPPoE [ses 0x1cb0] IP 118.250.6.85.64219 > dns2.cs.hn.cn.domain: UDP, len
23:43:05.811127 PPPoE [ses 0x1cb0] IP dns2.cs.hn.cn.domain > 118.250.6.85.64219: UDP, len
23:43:05.814764 IP dns2.cs.hn.cn.domain > 192.168.0.3.32804: UDP, length 42
23:43:05.816404 IP 192.168.0.3.32804 > dns2.cs.hn.cn.domain: UDP, length 42
23:43:05.816545 PPPoE [ses 0x1cb0] IP 118.250.6.85.64219 > dns2.cs.hn.cn.domain: UDP, len
10 packets captured
39 packets received by filter
0 packets dropped by kernel
```

转换克阅读格式

```
# tcpdump -d
(000) ret   #96
```

转换成十进制格式

```
# tcpdump -ddd
1
6 0 0 96
```

Linux cu命令

Linux cu命令用于连接另一个系统主机。

cu(call up)指令可连接另一台主机，并采用类似拨号终端机的接口工作，也可执行简易的文件传输作业。

语法

```
cu [dehnotv][-a<通信端口>][-c<电话号码>][-E<脱离字符>][-I<设置文件>][-l<外围设备代号>][-s<连线速率
```

参数说明：

- -a<通信端口>或-p<通信端口>或--port<通信端口> 使用指定的通信端口进行连线。
- -c<电话号码>或--phone<电话号码> 拨打该电话号码。
- -d 进入排错模式。
- -e或--parity=even 使用双同位检查。
- -E<脱离字符>或--escape<脱离字符> 设置脱离字符。
- -h或--halfduple 使用半双工模式。
- -I<配置文件>或--config<配置文件> 指定要使用的配置文件。
- -l<外围设备代号>或--line<外围设备代号> 指定某项外围设备，作为连接的设备。
- -n或--prompt 拨号时等待用户输入电话号码。
- -o或--parity=odd 使用单同位检查。
- -s<连线速率>或--speed<连线速率>或--baud<连线速率>或-<连线速率> 设置连线的速率，单位以鲍率计算。
- -t或--maper 把CR字符置换成LF+CR字符。
- -v或--version 显示版本信息。
- -x<排错模式>或--debug<排错模式> 使用排错模式。
- -z<系统主机>或--system<系统主机> 连接该系统主机。
- --help 在线帮助。
- --nostop 关闭Xon/Xoff软件流量控制。
- --parity=none 不使用同位检查。

实例

与远程主机连接

```
# cu -c 0102377765
```

Linux efax命令

Linux efax命令用于收发传真。

支持Class 1与Class 2的调制解调器来收发传真。

语法

```
efax [-sw][-a<AT指令>][-c<调制解调器属性>][-d<驱动程序>][-f<字体文件>][-g<指令>][-h<传真标题字符串
```

参数说明：

- -a<AT指令> 以指定的AT指令来接电话。
- -c<调制解调器属性> 设置本机调制解调器的属性。
- -d<驱动程序> 指定调制解调器驱动程序。
- -f<字体文件> 使用指定的字体文件来建立传真标题。
- -g<指令> 若接到的电话为数据，则执行指定的指令。
- -h<传真标题字符串> 指定字符串为每页最前端的标题。
- -i<AT指令> 在调制解调器进入传真模式前，传送AT指令到调制解调器。
- -j<AT指令> 在调制解调器进入传真模式后，传送AT指令到调制解调器。
- -k<AT指令> 在调制解调器离开传真模式前，传送AT指令到调制解调器。
- -l<识别码> 设置本机调制解调器的识别码。
- -o<选项> 使用非标准调制解调器时设置相关选项。
- -q<错误次数> 接收传真时，当每页发生错误次数超过指定的数目时，要求对方重发。
- -r<文件名> 在接收传真时，将每页分别保存成文件。
- -v<信息类型> 选择要印出的信息类型。
- -w 不要接听电话，等待OK或CONNECT的信号。
- -x<UUCP锁定文件> 使用UUCP格式的锁定文件来锁定调制解调器。
- -t<电话号码><传真文件> 以<电话号码>中的号码来拨号，并将<传真文件>传真出去。

Linux pppsetup命令

Linux pppsetup命令用于设置PPP连线。

这是Slackware发行版内附程序，它具有互动式的问答界面，让用户轻易完成PPP的连线设置。

语法

```
pppsetup
```

实例

设置ppp拨号

```
# pppsetup
```

Linux testparm命令

Linux testparm命令用于测试Samba的设置是否正确无误。

执行testparm(test parameter)指令可以简单测试Samba的配置文件，假如测试结果无误，Samba常驻服务就能正确载入该设置值，但并不保证其后的操作如预期般一切正常。

语法

```
testparm [-s][配置文件][<主机名称><IP地址>]
```

参数说明：

- -s 不显示提示符号等待用户按下Enter键，就直接列出Samba服务定义信息。

实例

查看Ssmba配置

```
# testparm
Load smb config files from /etc/samba/smb.conf
Processing section '[homes]'
Processing section '[printers]'
Processing section '[uptech]'
Processing section '[home]'
Loaded services file OK.
Server role: ROLE_STANDALONE
Press enter to see a dump of your service definitions
      ///按下回车继续
# Global parameters
[global]
workgroup = MYGROUP
server string = Samba Server
security = SHARE
encrypt passwords = No
password server = None
log file = /var/log/samba/%m.log
max log size = 50
socket options = TCP_NODELAY SO_RCVBUF=8192 SO_SNDBUF=8192
printcap name = /etc/printcap
dns proxy = No
idmap uid = 16777216-33554431
idmap gid = 16777216-33554431
cups options = raw

[homes]
comment = Home Directories
read only = No
browseable = No

[printers]
comment = All Printers
path = /var/spool/samba
printable = Yes
browseable = No

[uptech]
comment = *
path = /home/uptech
read only = No
guest ok = Yes

[home]
comment = *
path = /home
read only = No
guest ok = Yes
```

Linux smbclient命令

Linux smbclient命令可存取SMB/CIFS服务器的用户端程序。

SMB与CIFS为服务器通信协议，常用于Windows95/98/NT等系统。smbclient(samba client)可让Linux系统存取Windows系统所分享的资源。

语法

```
smbclient [网络资源][密码][-EhLN][-B<IP地址>][-d<排错层级>][-i<范围>][-I<IP地址>][-l<记录文件>][
```

参数说明：

- [网络资源] [网络资源]的格式为//服务器名称/资源分享名称。
- [密码] 输入存取网络资源所需的密码。
- -B<IP地址> 传送广播数据包时所用的IP地址。
- -d<排错层级> 指定记录文件所记载事件的详细程度。
- -E 将信息送到标准错误输出设备。
- -h 显示帮助。
- -i<范围> 设置NetBIOS名称范围。
- -I<IP地址> 指定服务器的IP地址。
- -l<记录文件> 指定记录文件的名称。
- -L 显示服务器端所分享出来的所有资源。
- -M<NetBIOS名称> 可利用WinPopup协议，将信息送给选项中所指定的主机。
- -n<NetBIOS名称> 指定用户端所要使用的NetBIOS名称。
- -N 不用询问密码。
- -O<连接槽选项> 设置用户端TCP连接槽的选项。
- -p<TCP连接端口> 指定服务器端TCP连接端口编号。
- -R<名称解析顺序> 设置NetBIOS名称解析的顺序。
- -s<目录> 指定smb.conf所在的目录。
- -t<服务器字码> 设置用何种字符码来解析服务器端的文件名称。
- -T<tar选项> 备份服务器端分享的全部文件，并打包成tar格式的文件。
- -U<用户名称> 指定用户名称。
- -W<工作群组> 指定工作群组名称。

Linux shapecfg命令

Linux shapecfg命令用于管制网络设备的流量。

自Linux-2.15开始，便支持流量管制的功能。

语法

```
shapecfg attach [流量管制器][网络设备]
```

或

```
shapecfg speed [流量管制器][带宽]
```

参数说明：

- attach 将流量管制器与实际的网络设备结合。
- speed 设置流量管制器的对外传输带宽。

Linux命令大全 - 系统管理

adduser	chfn	useradd	date
exit	finger	fwhios	sleep
suspend	groupdel	groupmod	halt
kill	last	lastb	login
logname	logout	ps	nice
procinfo	top	pstree	reboot
rlogin	rsh	sliplogin	screen
shutdown	rwho	sudo	gitps
swatch	tload	logrotate	uname
chsh	userconf	userdel	usermod
vlock	who	whoami	whois
newgrp	renice	su	skill
w	id	free	

Linux date命令

Linux date命令可以用来显示或设定系统的日期与时间，在显示方面，使用者可以设定欲显示的格式，格式设定为一个加号后接数个标记，其中可用的标记列表如下：

时间方面：

- % : 印出 %
- %n : 下一行
- %t : 跳格
- %H : 小时(00..23)
- %I : 小时(01..12)
- %k : 小时(0..23)
- %l : 小时(1..12)
- %M : 分钟(00..59)
- %p : 显示本地 AM 或 PM
- %r : 直接显示时间 (12 小时制，格式为 hh:mm:ss [AP]M)
- %s : 从 1970 年 1 月 1 日 00:00:00 UTC 到目前为止的秒数
- %S : 秒(00..61)
- %T : 直接显示时间 (24 小时制)
- %X : 相当于 %H:%M:%S
- %Z : 显示时区

日期方面：

- %a : 星期几 (Sun..Sat)
- %A : 星期几 (Sunday..Saturday)
- %b : 月份 (Jan..Dec)
- %B : 月份 (January..December)
- %c : 直接显示日期与时间
- %d : 日 (01..31)
- %D : 直接显示日期 (mm/dd/yy)
- %h : 同 %b
- %j : 一年中的第几天 (001..366)
- %m : 月份 (01..12)
- %U : 一年中的第几周 (00..53) (以 Sunday 为一周的第一天的情形)
- %w : 一周中的第几天 (0..6)
- %W : 一年中的第几周 (00..53) (以 Monday 为一周的第一天的情形)
- %x : 直接显示日期 (mm/dd/yy)
- %y : 年份的最后两位数字 (00.99)

- %Y : 完整年份 (0000..9999)

若是不以加号作为开头，则表示要设定时间，而时间格式为 MMDDhhmm[[CC]YY][.ss]，其中 MM 为月份，DD 为日，hh 为小时，mm 为分钟，CC 为年份前两位数字，YY 为年份后两位数字，ss 为秒数。

使用权限：所有使用者。

当您不希望出现无意义的 0 时(比如说 1999/03/07)，则可以在标记中插入 - 符号，比如说 date '+%-H:%-M:%-S' 会把时分秒中无意义的 0 给去掉，像是原本的 08:09:04 会变为 8:9:4。另外，只有取得权限者(比如说 root)才能设定系统时间。

当您以 root 身分更改了系统时间之后，请记得以 clock -w 来将系统时间写入 CMOS 中，这样下次重新开机时系统时间才会持续抱持最新的正确值。

语法

```
date [-u] [-d datestr] [-s datestr] [--utc] [--universal] [--date=datestr] [--set=datestr
```

参数说明：

- -d datestr : 显示 datestr 中所设定的时间 (非系统时间)
- --help : 显示辅助讯息
- -s datestr : 将系统时间设为 datestr 中所设定的时间
- -u : 显示目前的格林威治时间
- --version : 显示版本编号

实例

显示当前时间

```
# date
三 5月 12 14:08:12 CST 2010
# date '+%c'
2010年05月12日 星期三 14时09分02秒
# date '+%D' //显示完整的时间
05/12/10
# date '+%x' //显示数字日期，年份两位数表示
2010年05月12日
# date '+%T' //显示日期，年份用四位数表示
14:09:31
# date '+%X' //显示24小时的格式
14时09分39秒
```

按自己的格式输出

```
# date '+usr_time: $1:%M %P -hey'
usr_time: $1:16 下午 -hey
```

显示时间后跳行，再显示目前日期

```
date '+%T%n%D'
```

显示月份与日数

```
date '+%B %d'
```

显示日期与设定时间(12:34:56)

```
date --date '12:34:56'
```

Linux chfn命令

Linux chfn命令提供使用者更改个人资讯，用于 finger and mail username

使用权限：所有使用者。

语法

```
shell>> chfn
```

实例

改变finger信息

```
# chfn
Changing finger information for root.
Name [root]: hnlinux
Office []: hn
Office Phone []: 888888
Home Phone []: 9999999

Finger information changed.
```

改变账号真实姓名

```
# chfn -f hnunix
Changing finger information for root.
Finger information changed.
```

Linux adduser命令

Linux adduser命令用于新增使用者帐号或更新预设的使用者资料。

adduser 与 useradd 指令为同一指令（经由符号连结 symbolic link）。

使用权限：系统管理员。

adduser是增加使用者。相对的，也有删除使用者的指令，userdel。语法:userdel [login ID]

语法

```
adduser [-c comment] [-d home_dir] [-e expire_date] [-f inactive_time] [-g initial_group]
```

或

```
adduser -D [-g default_group] [-b default_home] [-f default_inactive] [-e default_expire_
```

参数说明：

- -c comment 新使用者位于密码档（通常是 /etc/passwd）的注解资料
- -d home_dir 设定使用者的家目录为 home_dir，预设值为预设的 home 后面加上使用者帐号 loginid
- -e expire_date 设定此帐号的使用期限（格式为 YYYY-MM-DD），预设值为永久有效
- -f inactive_time 范例：

实例

添加一个一般用户

```
# useradd kk //添加用户kk
```

为添加的用户指定相应的用户组

```
# useradd ?g root kk //添加用户kk，并指定用户所在的组为root用户组
```

创建一个系统用户

```
# useradd ?r kk //创建一个系统用户kk
```

为新添加的用户指定/home目录

```
# useradd-d /home/myf kk //新添加用户kk，其home目录为/home/myf
//当用户名kk登录主机时，系统进入的默认目录为/home/myf
```

Linux groupdel命令

Linux groupdel命令用于删除群组。

需要从系统上删除群组时，可用groupdel(group delete)指令来完成这项工作。倘若该群组中仍包括某些用户，则必须先删除这些用户后，方能删除群组。

语法

```
groupdel [群组名称]
```

实例

删除一个群组

```
# groupdel hnuser
```

Linux useradd命令

Linux useradd命令用于建立用户帐号。

useradd可用来建立用户帐号。帐号建好之后，再用passwd设定帐号的密码．而可用userdel删除帐号。使用useradd指令所建立的帐号，实际上是保存在/etc/passwd文本文件中。

语法

```
useradd [-mMnr][-c <备注>][-d <登入目录>][-e <有效期限>][-f <缓冲天数>][-g <群组>][-G <群组>][-
```

或

```
useradd -D [-b][-e <有效期限>][-f <缓冲天数>][-g <群组>][-G <群组>][-s <shell>]
```

参数说明：

- -c<备注>　加上备注文字。备注文字会保存在passwd的备注栏位中。
- -d<登入目录>　指定用户登入时的启始目录。
- -D　变更预设值．
- -e<有效期限>　指定帐号的有效期限。
- -f<缓冲天数>　指定在密码过期后多少天即关闭该帐号。
- -g<群组>　指定用户所属的群组。
- -G<群组>　指定用户所属的附加群组。
- -m　自动建立用户的登入目录。
- -M　不要自动建立用户的登入目录。
- -n　取消建立以用户名称为名的群组．
- -r　建立系统帐号。
- -s<shell>　指定用户登入后所使用的shell。
- -u<uid>　指定用户ID。

实例

添加一般用户

```
# useradd tt
```

为添加的用户指定相应的用户组

```
# useradd -g root tt
```

创建一个系统用户

```
# useradd -r tt
```

为新添加的用户指定home目录

```
# useradd -d /home/myd tt
```

建立用户且制定ID

```
# useradd caojh -u 544
```

Linux groupmod命令

Linux groupmod命令用于更改群组识别码或名称。

需要更改群组的识别码或名称时，可用groupmod指令来完成这项工作。

语法

```
groupmod [-g <群组识别码> <-o>][-n <新群组名称>][群组名称]
```

参数：

- -g <群组识别码>　设置欲使用的群组识别码。
- -o　重复使用群组识别码。
- -n <新群组名称>　设置欲使用的群组名称。

实例

修改组名

```
[root@w3cschool.cc ~]# groupadd linuxso
[root@w3cschool.cc ~]# tail -1 /etc/group
linuxso:x:500:
[root@w3cschool.cc ~]# tail -1 /etc/group
linuxso:x:500:
[root@w3cschool.cc ~]# groupmod -n linux linuxso
[root@w3cschool.cc ~]# tail -1 /etc/group
linux:x:500:
```

Linux logname命令

Linux logname命令用于显示用户名称。

执行logname指令，它会显示目前用户的名称。

语法

```
logname [--help][--version]
```

参数：

- --help 　在线帮助。
- --vesion 　显示版本信息。

实例

显示登录账号的信息：

```
# logname
root
```

Linux logout命令

Linux logout命令用于退出系统。

logout指令让用户退出系统，其功能和login指令相互对应。

语法

```
logout
```

实例

退出系统：

```
[root@w3cschool.cc ~]# logout
```

Linux ps命令

Linux ps命令用于显示当前进程 (process) 的状态。

语法

```
ps [options] [--help]
```

参数：

- ps 的参数非常多, 在此仅列出几个常用的参数并大略介绍含义
- -A 列出所有的行程
- -w 显示加宽可以显示较多的资讯
- -au 显示较详细的资讯
- -aux 显示所有包含其他使用者的行程
- au(x) 输出格式 :
- USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
- USER: 行程拥有者
- PID: pid
- %CPU: 占用的 CPU 使用率
- %MEM: 占用的记忆体使用率
- VSZ: 占用的虚拟记忆体大小
- RSS: 占用的记忆体大小
- TTY: 终端的次要装置号码 (minor device number of tty)
- STAT: 该行程的状态:
- D: 不可中断的静止 (通悸□□縝b进行 I/O 动作)
- R: 正在执行中
- S: 静止状态
- T: 暂停执行
- Z: 不存在但暂时无法消除
- W: 没有足够的记忆体分页可分配
- <: 高优先序的行程
- N: 低优先序的行程
- L: 有记忆体分页分配并锁在记忆体内 (实时系统或捱A I/O)
- START: 行程开始时间
- TIME: 执行的时间
- COMMAND:所执行的指令

实例

```
# ps -A 显示进程信息
PID TTY      TIME CMD
  1 ?    00:00:02 init
  2 ?    00:00:00 kthreadd
  3 ?    00:00:00 migration/0
  4 ?    00:00:00 ksoftirqd/0
  5 ?    00:00:00 watchdog/0
  6 ?    00:00:00 events/0
  7 ?    00:00:00 cpuset
  8 ?    00:00:00 khelper
  9 ?    00:00:00 netns
  10 ?    00:00:00 async/mgr
  11 ?    00:00:00 pm
  12 ?    00:00:00 sync_supers
  13 ?    00:00:00 bdi-default
  14 ?    00:00:00 kintegrityd/0
  15 ?    00:00:02 kblockd/0
  16 ?    00:00:00 kacpid
  17 ?    00:00:00 kacpi_notify
  18 ?    00:00:00 kacpi_hotplug
  19 ?    00:00:27 ata/0
……省略部分结果
30749 pts/0  00:00:15 gedit
30886 ?    00:01:10 qtcreator.bin
30894 ?    00:00:00 qtcreator.bin
31160 ?    00:00:00 dhclient
31211 ?    00:00:00 aptd
31302 ?    00:00:00 sshd
31374 pts/2  00:00:00 bash
31396 pts/2  00:00:00 ps
```

显示指定用户信息

```
# ps -u root //显示root进程用户信息
 PID TTY      TIME CMD
  1 ?    00:00:02 init
  2 ?    00:00:00 kthreadd
  3 ?    00:00:00 migration/0
  4 ?    00:00:00 ksoftirqd/0
  5 ?    00:00:00 watchdog/0
  6 ?    00:00:00 events/0
  7 ?    00:00:00 cpuset
  8 ?    00:00:00 khelper
  9 ?    00:00:00 netns
  10 ?    00:00:00 async/mgr
  11 ?    00:00:00 pm
  12 ?    00:00:00 sync_supers
  13 ?    00:00:00 bdi-default
  14 ?    00:00:00 kintegrityd/0
  15 ?    00:00:02 kblockd/0
  16 ?    00:00:00 kacpid
……省略部分结果
30487 ?    00:00:06 gnome-terminal
30488 ?    00:00:00 gnome-pty-helpe
30489 pts/0  00:00:00 bash
30670 ?    00:00:00 debconf-communi
30749 pts/0  00:00:15 gedit
30886 ?    00:01:10 qtcreator.bin
30894 ?    00:00:00 qtcreator.bin
31160 ?    00:00:00 dhclient
31211 ?    00:00:00 aptd
31302 ?    00:00:00 sshd
31374 pts/2  00:00:00 bash
31397 pts/2  00:00:00 ps
```

显示所有进程信息，连同命令行

```
# ps -ef //显示所有命令，连带命令行
UID    PID PPID C STIME TTY     TIME CMD
root     1   0 0 10:22 ?    00:00:02 /sbin/init
root     2   0 0 10:22 ?    00:00:00 [kthreadd]
root     3   2 0 10:22 ?    00:00:00 [migration/0]
root     4   2 0 10:22 ?    00:00:00 [ksoftirqd/0]
root     5   2 0 10:22 ?    00:00:00 [watchdog/0]
root     6   2 0 10:22 ?    /usr/lib/NetworkManager
……省略部分结果
root   31302 2095 0 17:42 ?    00:00:00 sshd: root@pts/2
root   31374 31302 0 17:42 pts/2  00:00:00 -bash
root   31400   1 0 17:46 ?    00:00:00 /usr/bin/python /usr/sbin/aptd
root   31407 31374 0 17:48 pts/2  00:00:00 ps -ef
```

Linux exit命令

Linux exit命令用于退出目前的shell。

执行exit可使shell以指定的状态值退出。若不设置状态值参数，则shell以预设值退出。状态值0代表执行成功，其他值代表执行失败。exit也可用在script，离开正在执行的script，回到shell。

语法

```
exit [状态值]
```

实例

退出终端

```
# exit
```

Linux finger命令

Linux finger命令可以让使用者查询一些其他使用者的资料。会列出来的资料有：

- Login Name
- User Name
- Home directory
- Shell
- Login status
- mail status
- .plan
- .project
- .forward

其中 .plan、.project 和 .forward 就是使用者在他的 Home Directory 里的 .plan， .project 和 .forward 等档案里的资料。如果没有就没有。finger 指令并不限定于在同一服务器上查询，也可以寻找某一个远端服务器上的使用者。只要给一个像是 E-mail address 一般的地址即可。

使用权限：所有使用者。

语法

```
finger [options] user[@address]
```

参数说明：

- -l 多行显示。
- -s 单行显示。这个选项只显示登入名称、真实姓名、终端机名称、闲置时间、登入时间、办公室号码及电话号码。如果所查询的使用者是远端服务器的使用者，这个选项无效。

实例

列出当前登录用户的相关信息

```
# finger -l //显示用户信息
Login: root Name: root
Directory: /root Shell: /bin/bash
On since Fri Apr 9 20:17 (CST) on :0 (messages off)
On since Fri Apr 9 20:17 (CST) on pts/1 32 days 22 hours idle
On since Fri Apr 9 20:17 (CST) on pts/3 4 hours 5 minutes idle
(messages off)
On since Wed May 12 18:08 (CST) on pts/4 from 192.168.1.10
On since Wed May 12 18:35 (CST) on pts/5 from 192.168.1.10
7 minutes 54 seconds idle
On since Wed May 12 14:37 (CST) on pts/2 from 192.168.1.10
3 hours 14 minutes idle
On since Wed May 12 14:53 (CST) on pts/7 34 minutes 25 seconds idle
(messages off)
On since Wed May 12 16:53 (CST) on pts/8 from 192.168.1.10
30 minutes 18 seconds idle
Mail last read Mon Mar 31 04:02 2008 (CST)
No Plan.
```

显示指定用户信息

```
# finger -m hnlinux
```

显示远程用户信息

```
# finger -m root@192.168.1.13
```

下列指令可以查询本机管理员的资料：

```
finger root
```

其结果如下：

```
Login: root Name: root
Directory: /root Shell: /bin/bash
Never logged in.
No mail.
No Plan.
```

Linux fwhios命令

Linux fwhios命令用于查找并显示用户信息。

本指令的功能有点类似finger指令，它会去查找并显示指定帐号的用户相关信息。不同之处在于fwhois指令是到Network Solutions的WHOIS数据库去查找，该帐号名称必须有在上面注册才能寻获，且名称没有大小写的差别。

语法

```
fwhios [帐号名称]
```

Linux sleep命令

Linux sleep命令可以用来将目前动作延迟一段时间。

使用权限：所有使用者。

语法

```
sleep [--help] [--version] number[smhd]
```

参数说明：

- --help : 显示辅助讯息
- --version : 显示版本编号
- number : 时间长度，后面可接 s、m、h 或 d
- 其中 s 为秒，m 为 分钟，h 为小时，d 为日数

实例

休眠5分钟

```
# sleep 5m
```

显示目前时间后延迟 1 分钟，之后再次显示时间

```
date;sleep 1m;date
```

Linux suspend命令

Linux suspend命令用于暂停执行shell。

suspend为shell内建指令，可暂停目前正在执行的shell。若要恢复，则必须使用SIGCONT信息。

语法

```
suspend [-f]
```

参数说明：

- -f 若目前执行的shell为登入的shell，则suspend预设无法暂停此shell。若要强迫暂停登入的shell，则必须使用-f参数。

实例

暂停shell

```
# suspend
-bash: suspend: 无法挂起一个登录 shell
# suspend -f
```

Linux login命令

Linux login命令用于登入系统。

login指令让用户登入系统，您亦可通过它的功能随时更换登入身份。在Slackware发行版中，您可在指令后面附加欲登入的用户名称，它会直接询问密码，等待用户输入。当/etc目录里含名称为nologin的文件时，系统只root帐号登入系统，其他用户一律不准登入。

语法

```
login
```

实例

使用新的身份登录系统

```
# login
```

Linux lastb命令

Linux lastb命令用于列出登入系统失败的用户相关信息。

单独执行lastb指令，它会读取位于/var/log目录下，名称为btmp的文件，并把该文件内容记录的登入失败的用户名单，全部显示出来。

语法

```
lastb [-adRx][-f <记录文件>][-n <显示列数>][帐号名称...][终端机编号...]
```

参数说明：

- -a 　把从何处登入系统的主机名称或IP地址显示在最后一行。
- -d 　将IP地址转换成主机名称。
- -f<记录文件> 　指定记录文件。
- -n<显示列数>或-<显示列数> 　设置列出名单的显示列数。
- -R 　不显示登入系统的主机名称或IP地址。
- -x 　显示系统关机，重新开机，以及执行等级的改变等信息。

实例

显示登录失败的用户

```
# lastb
root    tty7      :1          Thu May 13 11:26 - 11:26 (00:00)

btmp begins Thu May 13 11:26:39 2014
```

Linux rlogin命令

Linux rlogin命令用于远端登入。

执行rlogin指令开启终端机阶段操作，并登入远端主机。

语法

```
rlogin [-8EL][-e <脱离字符>][-l <用户名称>][主机名称或IP地址]
```

必要参数：

- -E 忽略escape字符
- -8 只识别8位字的字符
- -L 允许rlogin会话运行在litout模式
- -ec 设置escape字符为c
- -c 断开连接前要求确认
- -a 强制要求远程主机在发送完一个空的本地用户名之后请求一个密码
- -f 向远端主机发送一个本地认证
- -F 向远程主机发送一个可转寄的本地认证
- -7 强制执行7为的传输
- -d 打开用于远端主机通信的TCP套接口的调试
- -k 要求包含远端主机的tisckets
- -x 启动数据传输的DES加密
- -4 只使用 kerkberos的版本4的认证

选择参数：

- -e<字符> 设置退出字符 -l<用户> 指定登陆的用户 -t<终端类型> 设置终端类型

实例

显示rlogin服务是否开启

```
# chkconfig --list //检测rlogin服务是否开启
```

开启rlogin服务

```
# chkconfig rlogin on //开启rlogin服务
```

登陆远程主机

```
# rlogin 192.168.1.88
Password：
Password：
Login incorrect
Login:root
Passwd:
Login incorrect
Login:kk
Passwd:
```

指定用户名登陆远程主机

```
# rlogin 192.168.1.88 -l hnlinux

Passord:
Last login：Mon May 28 15：30:25 from 192.168.1.88

#
```

Linux last命令

Linux last命令用于显示系统开机以来获是从每月初登入者的讯息。

使用权限：所有使用者。

语法

```
shell>> last [options]
```

参数说明：

- -R 省略 hostname 的栏位
- -num 展示前 num 个
- username 展示 username 的登入讯息
- tty 限制登入讯息包含终端机代号

实例

```
shell>> last -R -2
johnney pts/1 Mon Aug 14 20:42 still logged in
johnney pts/0 Mon Aug 14 19:59 still logged in
wtmp begins Tue Aug 1 09:01:10 2000 ### /var/log/wtmp
shell>> last -2 minery
minery pts/0 140.119.217.115 Mon Aug 14 18:37 - 18:40 (00:03)
minery pts/0 140.119.217.115 Mon Aug 14 17:22 - 17:24 (00:02)
wtmp begins Tue Aug 1 09:01:10 2000
```

一般显示方法

```
# last
```

简略显示，并指定显示的个数

```
# last -n 5 -R
root    pts/4    Thu May 13 17:25  still logged in
root    pts/2    Thu May 13 17:23 - 17:25 (00:02)
root    pts/1    Thu May 13 16:46  still logged in
root    pts/7    Thu May 13 15:36  still logged in
root    pts/9    Thu May 13 15:35  still logged in

wtmp begins Thu May 13 18:55:40 2014
```

显示最后一列显示主机IP地址

```
# last -n 5 -a -i
root   pts/4    Thu May 13 17:25  still logged in  192.168.1.10
root   pts/2    Thu May 13 17:23 - 17:25 (00:02)   192.168.1.10
root   pts/1    Thu May 13 16:46  still logged in  192.168.1.10
root   pts/7    Thu May 13 15:36  still logged in  192.168.1.10
root   pts/9    Thu May 13 15:35  still logged in  192.168.1.10

wtmp begins Thu May 13 18:55:40 2014
```

Linux reboot命令

Linux reboot命令用于用来重新启动计算机。

若系统的 runlevel 为 0 或 6 ，则重新开机，否则以 shutdown 指令（加上 -r 参数）来取代

语法

```
reboot [-n] [-w] [-d] [-f] [-i]
```

参数：

- -n : 在重开机前不做将记忆体资料写回硬盘的动作
- -w : 并不会真的重开机，只是把记录写到 /var/log/wtmp 档案里
- -d : 不把记录写到 /var/log/wtmp 档案里（-n 这个参数包含了 -d）
- -f : 强迫重开机，不呼叫 shutdown 这个指令
- -i : 在重开机之前先把所有网络相关的装置先停止

实例

重新启动

```
# reboot
```

Linux kill命令

Linux kill命令用于删除执行中的程序或工作。

kill可将指定的信息送至程序。预设的信息为SIGTERM(15)，可将指定程序终止。若仍无法终止该程序，可使用SIGKILL(9)信息尝试强制删除程序。程序或工作的编号可利用ps指令或jobs指令查看。

语法

```
kill [-s <信息名称或编号>][程序]  或  kill [-l <信息编号>]
```

参数说明：

- -l <信息编号>　若不加<信息编号>选项，则-l参数会列出全部的信息名称。
- -s <信息名称或编号>　指定要送出的信息。
- [程序]　[程序]可以是程序的PID或是PGID，也可以是工作编号。

实例

杀死进程

```
# kill 12345
```

强制杀死进程

```
# kill -KILL 123456
```

发送SIGHUP信号，可以使用一下信号

```
# kill -HUP pid
```

彻底杀死进程

```
# kill -9 123456
```

显示信号

```
# kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
6) SIGABRT      7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT     17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG     24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM     27) SIGPROF     28) SIGWINCH     29) SIGIO     30) SIGPWR
31) SIGSYS     34) SIGRTMIN     35) SIGRTMIN+1     36) SIGRTMIN+2     37) SIGRTMIN+3
38) SIGRTMIN+4     39) SIGRTMIN+5     40) SIGRTMIN+6     41) SIGRTMIN+7     42) SIGRTMIN+8
43) SIGRTMIN+9     44) SIGRTMIN+10     45) SIGRTMIN+11     46) SIGRTMIN+12     47) SIGRTMIN+1
48) SIGRTMIN+14     49) SIGRTMIN+15     50) SIGRTMAX-14     51) SIGRTMAX-13     52) SIGRTMAX-
53) SIGRTMAX-11     54) SIGRTMAX-10     55) SIGRTMAX-9     56) SIGRTMAX-8     57) SIGRTMAX-7
58) SIGRTMAX-6     59) SIGRTMAX-5     60) SIGRTMAX-4     61) SIGRTMAX-3     62) SIGRTMAX-2
63) SIGRTMAX-1     64) SIGRTMAX
```

杀死指定用户所有进程

```
#kill -9 $(ps -ef | grep hnlinux) //方法一 过滤出hnlinux用户进程
#kill -u hnlinux //方法二
```

Linux halt命令

若系统的 runlevel 为 0 或 6 ，则Linux halt命令关闭系统，否则以 shutdown 指令（加上 -h 参数）来取代。

使用权限：系统管理者。

语法

```
halt [-n] [-w] [-d] [-f] [-i] [-p]
```

参数说明：

- -n : 在关机前不做将记忆体资料写回硬盘的动作
- -w : 并不会真的关机，只是把记录写到 /var/log/wtmp 档案里
- -d : 不把记录写到 /var/log/wtmp 档案里（-n 这个参数包含了 -d） -f : 强迫关机，不呼叫 shutdown 这个指令
- -i : 在关机之前先把所有网络相关的装置先停止
- -p : 当关机的时候，顺便做关闭电源（poweroff）的动作

实例

关闭系统

```
# halt
```

关闭系统并关闭电源

```
# halt -p
```

关闭系统，但不留下纪录

```
# halt -d
```

Linux nice命令

Linux nice命令以更改过的优先序来执行程序，如果未指定程序，则会印出目前的排程优先序，内定的 adjustment 为 10，范围为 -20（最高优先序）到 19（最低优先序）。

使用权限：所有使用者。

语法

```
nice [-n adjustment] [-adjustment] [--adjustment=adjustment] [--help] [--version] [comman
```

参数说明：

- -n adjustment, -adjustment, --adjustment=adjustment 皆为将该原有优先序的增加 adjustment
- --help 显示求助讯息
- --version 显示版本资讯

实例

设置程序运行时的优先级

```
# vi & //后台运行
[1] 15297
# nice vi & //设置默认优先级
[2] 15298

[1]+ Stopped         vi
# nice -n 19 vi & //设置优先级为19
[3] 15299

[2]+ Stopped         nice vi
# nice -n -20 vi & //设置优先级为 -20
[4] 15300

[3]+ Stopped         nice -n 19 vi
# ps -l //显示进程
F S  UID  PID PPID C PRI NI ADDR SZ WCHAN TTY     TIME CMD
4 S   0 15278 15212 0 80  0 - 1208 wait  pts/2  00:00:00 bash
0 T   0 15297 15278 0 80  0 - 2687 signal pts/2  00:00:00 vi
0 T   0 15298 15278 0 90 10 - 2687 signal pts/2  00:00:00 vi
0 T   0 15299 15278 1 99 19 - 2687 signal pts/2  00:00:00 vi
4 T   0 15300 15278 3 60 -20 - 2687 signal pts/2  00:00:00 vi
4 R   0 15301 15278 0 80  0 -  625 -   pts/2  00:00:00 ps

[4]+ Stopped         nice -n -20 vi
```

将 ls 的优先序加 1 并执行

```
nice -n 1 ls
```

将 ls 的优先序加 10 并执行

```
nice ls
```

注意：优先序 (priority) 为操作系统用来决定 CPU 分配的参数，Linux 使用『回合制(round-robin)』的演算法来做 CPU 排程，优先序越高，所可能获得的 CPU时间就越多。

Linux procinfo命令

Linux procinfo命令用于显示系统状态。

procinfo(process information)指令从/proc目录里读取相关数据，将数据妥善整理过后输出到标准输出设备。

语法

```
procinfo [-abdDfhimsSv][-F <输出文件>][-n <间隔秒数>]
```

参数说明：

- -a 　显示所有信息。
- -b 　显示磁盘设备的区块数目，而非存取数目。
- -d 　显示系统信息每秒间的变化差额，而非总和的数值。本参数必须配合"-f"参数使用
- -D 　此参数效果和指定"-d"参数类似，但内存和交换文件的信息为总和数值。
- -f 　进入全画面的互动式操作界面。
- -F<输出文件> 　把信息状态输出到文件保存起来，而非预设的标准输出设备。
- -h 　在线帮助。
- -i 　显示完整的IRP列表。
- -m 　显示系统模块和外围设备等相关信息。
- -n<间隔秒数> 　设置全画面互动模式的信息更新速度，单位以秒计算。
- -s 　显示系统的内存，磁盘空间，IRP和DMA等信息，此为预设值。
- -S 　搭配参数"-d"或"-D"使用时，每秒都会更新信息，不论是否有使用参数"-n"。
- -v 　显示版本信息。

实例

显示系统状态

```
# procinfo
```

Linux top命令

Linux top命令用于实时显示 process 的动态。

使用权限：所有使用者。

语法

```
top [-] [d delay] [q] [c] [S] [s] [i] [n] [b]
```

参数说明：

- d : 改变显示的更新速度，或是在交谈式指令列(interactive command)按 s
- q : 没有任何延迟的显示速度，如果使用者是有 superuser 的权限，则 top 将会以最高的优先序执行
- c : 切换显示模式，共有两种模式，一是只显示执行档的名称，另一种是显示完整的路径与名称S : 累积模式，会将己完成或消失的子行程 (dead child process) 的 CPU time 累积起来
- s : 安全模式，将交谈式指令取消, 避免潜在的危机
- i : 不显示任何闲置 (idle) 或无用 (zombie) 的行程
- n : 更新的次数，完成后将会退出 top
- b : 批次档模式，搭配 "n" 参数一起使用，可以用来将 top 的结果输出到档案内

实例

显示进程信息

```
# top
```

显示完整命令

```
# top -c
```

以批处理模式显示程序信息

```
# top -b
```

以累积模式显示程序信息

```
# top -S
```

设置信息更新次数

```
top -n 2
//表示更新两次后终止更新显示
```

设置信息更新时间

```
# top -d 3
//表示更新周期为3秒
```

显示指定的进程信息

```
# top -p 139
//显示进程号为139的进程信息，CPU、内存占用率等
```

显示更新十次后退出

```
top -n 10
```

使用者将不能利用交谈式指令来对行程下命令

```
top -s
```

将更新显示二次的结果输入到名称为 top.log 的档案里

```
top -n 2 -b < top.log
```

Linux pstree命令

Linux pstree命令将所有行程以树状图显示，树状图将会以 pid (如果有指定) 或是以 init 这个基本行程为根 (root)，如果有指定使用者 id，则树状图会只显示该使用者所拥有的行程。

使用权限：所有使用者。

语法

```
pstree [-a] [-c] [-h|-Hpid] [-l] [-n] [-p] [-u] [-G|-U] [pid|user]
```

或

```
pstree -V
```

参数说明：

- -a 显示该行程的完整指令及参数, 如果是被记忆体置换出去的行程则会加上括号
- -c 如果有重覆的行程名, 则分开列出（预设值是会在前面加上 *）

实例

显示进程的关系

```
pstree
init-+-amd
|-apmd
|-atd
|-httpd---10*[httpd]
%pstree -p
init(1)-+-amd(447)
|-apmd(105)
|-atd(339)
%pstree -c
init-+-amd
|-apmd
|-atd
|-httpd-+-httpd
| |-httpd
| |-httpd
| |-httpd
....
```

特别表明在运行的进程

```
# pstree -apnh //显示进程间的关系
```

同时显示用户名称

```
# pstree -u //显示用户名称
```

Linux shutdown命令

Linux shutdown命令可以用来进行关机程序，并且在关机以前传送讯息给所有使用者正在执行的程序，shutdown 也可以用来重开机。

使用权限：系统管理者。

语法

```
shutdown [-t seconds] [-rkhncfF] time [message]
```

参数说明：

- -t seconds : 设定在几秒钟之后进行关机程序
- -k : 并不会真的关机，只是将警告讯息传送给所有只用者
- -r : 关机后重新开机
- -h : 关机后停机
- -n : 不采用正常程序来关机，用强迫的方式杀掉所有执行中的程序后自行关机
- -c : 取消目前已经进行中的关机动作
- -f : 关机时，不做 fcsk 动作(检查 Linux 档系统)
- -F : 关机时，强迫进行 fsck 动作
- time : 设定关机的时间
- message : 传送给所有使用者的警告讯息

实例

立即关机

```
# shutdown -h now
```

指定5分钟后关机

```
# shutdown +5 “System will shutdown after 5 minutes” //5分钟够关机并显示警告信息
```

Linux screen命令

Linux screen命令用于多重视窗管理程序。

screen为多重视窗管理程序。此处所谓的视窗，是指一个全屏幕的文字模式画面。通常只有在使用telnet登入主机或是使用老式的终端机时，才有可能用到screen程序。

语法

```
screen [-AmRvx -ls -wipe][-d <作业名称>][-h <行数>][-r <作业名称>][-s <shell>][-S <作业名称>]
```

参数说明：

- -A 将所有的视窗都调整为目前终端机的大小。
- -d<作业名称> 将指定的screen作业离线。
- -h<行数> 指定视窗的缓冲区行数。
- -m 即使目前已在作业中的screen作业，仍强制建立新的screen作业。
- -r<作业名称> 恢复离线的screen作业。
- -R 先试图恢复离线的作业。若找不到离线的作业，即建立新的screen作业。
- -s<shell> 指定建立新视窗时，所要执行的shell。
- -S<作业名称> 指定screen作业的名称。
- -v 显示版本信息。
- -x 恢复之前离线的screen作业。
- -ls或--list 显示目前所有的screen作业。
- -wipe 检查目前所有的screen作业，并删除已经无法使用的screen作业。

实例

创建 screen 终端

```
# screen //创建 screen 终端
```

创建 screen 终端 并执行任务

```
# screen vi ~/main.c //创建 screen 终端 ，并执行 vi命令
```

离开 screen 终端

```
# screen vi ~/main.c //创建 screen 终端 ，并执行 vi命令

#include

main ()
{

}

"~/mail.c"       0,0-1

在 screen 终端 下 按下 Ctrl+a d键
```

重新连接离开的 screen 终端

```
# screen -ls  //显示已创建的screen终端
There are screens on:
2433.pts-3.linux    (2013年10月20日 16时48分59秒)    (Detached)
2428.pts-3.linux    (2013年10月20日 16时48分05秒)    (Detached)
2284.pts-3.linux    (2013年10月20日 16时14分55秒)    (Detached)
2276.pts-3.linux    (2013年10月20日 16时13分18秒)    (Detached)
4 Sockets in /var/run/screen/S-root.

# screen -r 2276 //连接 screen_id 为 2276 的 screen终端
```

Linux sliplogin命令

Linux sliplogin命令用于将SLIP接口加入标准输入。

sliplogin可将SLIP接口加入标准输入，把一般终端机的连线变成SLIP连线。通常可用来建立SLIP服务器，让远端电脑以SLIP连线到服务器。sliplogin活去检查/etc/slip/slip.hosts文件中是否有相同的用户名称。通过检查后，sliplogin会调用执行shell script来设置IP地址，子网掩码等网络界面环境。此shell script通常是/etc/slip/slip.login。

语法

```
sliplogin [用户名称]
```

实例

改变用户的连接方式

```
# sliplogin kk // 改变用户的连接方式
```

Linux rsh命令

Linux rsh命令用于远端登入的Shell。

rsh(remote shell)提供用户环境，也就是Shell，以便指令能够在指定的远端主机上执行。

语法

```
rsh [-dn][-l <用户名称>][主机名称或IP地址][执行指令]
```

参数说明：

- -d 　使用Socket层级的排错功能。
- -l<用户名称> 　指定要登入远端主机的用户名称。
- -n 　把输入的指令号向代号为/dev/null的特殊外围设备。

实例

开启rsh服务

```
# chkconfig --list //检测rlogin服务是否开启

# chkconfig rsh on //开启rsh服务

# chkconfig -list //检测开启的服务
```

远程命令执行

```
# rsh -l hnlinux 192.168.1.88 /bin/ls //远程执行ls命令
```

Linux rwho命令

Linux rwho命令用于查看系统用户。

rwho指令的效果类似who指令，但它会显示局域网里所有主机的用户。主机必须提供rwhod常驻服务的功能，方可使用rwho指令。

语法

```
rwho [-a]
```

参数说明：

- -a 列出所有的用户，包括闲置时间超过1个小时以上的用户。

实例

显示本地局域网内的所有用户

```
# rwho
root   snail-hnlinux:pts/2 May 14 17:42
```

Linux sudo命令

Linux sudo命令以系统管理者的身份执行指令，也就是说，经由 sudo 所执行的指令就好像是 root 亲自执行。

使用权限：在 /etc/sudoers 中有出现的使用者。

语法

```
sudo -V
```

```
sudo -h
```

```
sudo -l
```

```
sudo -v
```

```
sudo -k
```

```
sudo -s
```

```
sudo -H
```

```
sudo [ -b ] [ -p prompt ] [ -u username/#uid] -s
```

```
sudo command
```

参数说明：

- -V 显示版本编号
- -h 会显示版本编号及指令的使用方式说明
- -l 显示出自己（执行 sudo 的使用者）的权限
- -v 因为 sudo 在第一次执行时或是在 N 分钟内没有执行（N 预设为五）会问密码，这个参数是重新做一次确认，如果超过 N 分钟，也会问密码
- -k 将会强迫使用者在下一次执行 sudo 时问密码（不论有没有超过 N 分钟）
- -b 将要执行的指令放在背景执行

- -p prompt 可以更改问密码的提示语，其中 %u 会代换为使用者的帐号名称， %h 会显示主机名称
- -u username/#uid 不加此参数，代表要以 root 的身份执行指令，而加了此参数，可以以 username 的身份执行指令（#uid 为该 username 的使用者号码）
- -s 执行环境变数中的 SHELL 所指定的 shell ，或是 /etc/passwd 里所指定的 shell
- -H 将环境变数中的 HOME （家目录）指定为要变更身份的使用者家目录（如不加 -u 参数就是系统管理者 root ）
- command 要以系统管理者身份（或以 -u 更改为其他人）执行的指令

实例

sudo命令使用

```
$ sudo ls
[sudo] password for hnlinux:
hnlinux is not in the sudoers file. This incident will be reported.
```

指定用户执行命令

```
# sudo -u userb ls -l
```

显示sudo设置

```
$ sudo -L //显示sudo设置
Available options in a sudoers ``Defaults'' line:

syslog: Syslog facility if syslog is being used for logging
syslog_goodpri: Syslog priority to use when user authenticates successfully
syslog_badpri: Syslog priority to use when user authenticates unsuccessfully
long_otp_prompt: Put OTP prompt on its own line
ignore_dot: Ignore '.' in $PATH
mail_always: Always send mail when sudo is run
mail_badpass: Send mail if user authentication fails
mail_no_user: Send mail if the user is not in sudoers
mail_no_host: Send mail if the user is not in sudoers for this host
mail_no_perms: Send mail if the user is not allowed to run a command
tty_tickets: Use a separate timestamp for each user/tty combo
lecture: Lecture user the first time they run sudo
lecture_file: File containing the sudo lecture
authenticate: Require users to authenticate by default
root_sudo: Root may run sudo
log_host: Log the hostname in the (non-syslog) log file
log_year: Log the year in the (non-syslog) log file
shell_noargs: If sudo is invoked with no arguments, start a shell
set_home: Set $HOME to the target user when starting a shell with -s
always_set_home: Always set $HOME to the target user's home directory
path_info: Allow some information gathering to give useful error messages
fqdn: Require fully-qualified hostnames in the sudoers file
insults: Insult the user when they enter an incorrect password
requiretty: Only allow the user to run sudo if they have a tty
env_editor: Visudo will honor the EDITOR environment variable
rootpw: Prompt for root's password, not the users's
runaspw: Prompt for the runas_default user's password, not the users's
targetpw: Prompt for the target user's password, not the users's
use_loginclass: Apply defaults in the target user's login class if there is one
set_logname: Set the LOGNAME and USER environment variables
```

```
stay_setuid: Only set the effective uid to the target user, not the real uid
preserve_groups: Don't initialize the group vector to that of the target user
loglinelen: Length at which to wrap log file lines (0 for no wrap)
timestamp_timeout: Authentication timestamp timeout
passwd_timeout: Password prompt timeout
passwd_tries: Number of tries to enter a password
umask: Umask to use or 0777 to use user's
logfile: Path to log file
mailerpath: Path to mail program
mailerflags: Flags for mail program
mailto: Address to send mail to
mailfrom: Address to send mail from
mailsub: Subject line for mail messages
badpass_message: Incorrect password message
timestampdir: Path to authentication timestamp dir
timestampowner: Owner of the authentication timestamp dir
exempt_group: Users in this group are exempt from password and PATH requirements
passprompt: Default password prompt
passprompt_override: If set, passprompt will override system prompt in all cases.
runas_default: Default user to run commands as
secure_path: Value to override user's $PATH with
editor: Path to the editor for use by visudo
listpw: When to require a password for 'list' pseudocommand
verifypw: When to require a password for 'verify' pseudocommand
noexec: Preload the dummy exec functions contained in 'noexec_file'
noexec_file: File containing dummy exec functions
ignore_local_sudoers: If LDAP directory is up, do we ignore local sudoers file
closefrom: File descriptors >= %d will be closed before executing a command
closefrom_override: If set, users may override the value of `closefrom' with the -C optio
setenv: Allow users to set arbitrary environment variables
env_reset: Reset the environment to a default set of variables
env_check: Environment variables to check for sanity
env_delete: Environment variables to remove
env_keep: Environment variables to preserve
role: SELinux role to use in the new security context
type: SELinux type to use in the new security context
askpass: Path to the askpass helper program
env_file: Path to the sudo-specific environment file
sudoers_locale: Locale to use while parsing sudoers
visiblepw: Allow sudo to prompt for a password even if it would be visisble
pwfeedback: Provide visual feedback at the password prompt when there is user input
fast_glob: Use faster globbing that is less accurate but does not access the filesystem
umask_override: The umask specified in sudoers will override the user's, even if it is mo
```

以root权限执行上一条命令

```
$ sudo !!
```

以特定用户身份进行编辑文本

```
$ sudo -u uggc vi ~www/index.html
//以 uggc 用户身份编辑  home 目录下www目录中的 index.html 文件
```

列出目前的权限

```
sudo -l
```

列出 sudo 的版本资讯

```
sudo -V
```

Linux gitps命令

Linux gitps命令用于报告程序状况。

gitps(gnu interactive tools process status)是用来报告并管理程序执行的指令，基本上它就是通过ps指令来报告，管理程序，也能通过gitps指令随时中断，删除不必要的程序。因为gitps指令会去执行ps指令，所以其参数和ps指令相当类似。

语法

```
gitps [acefgjlnrsSTuvwxX][p <程序识别码>][t <终端机编号>][U <帐号名称>]
```

参数说明：

- a 　显示　现行终端机下的所有程序，包括其他用户的程序。
- c 　列出程序时，显示每个程序真正的指令名称，而不包含路径，参数或是常驻服务的标示.
- e 　列出程序时，显示每个程序所使用的环境变量。
- f 　用ASCII字符显示树状结构，表达程序间的相互关系。
- g 　显示现行终端机下的所有程序，包括群组领导者的程序。
- j 　采用工作控制的格式来显示程序状况。
- l 　采用纤细的格式来显示程序状况。
- n 　以数字来表示USER和WCHAN栏位。
- p<程序识别码> 　指定程序识别码，并列出该程序的状况。
- r 　只列出现行终端机正在执行中的程序。
- s 　采用程序信号的格式显示程序状况。
- S 　列出程序时，包括已中断的子程序信息。
- t<终端机机标号> 　指定终端机编号，并列出属于该终端机的程序的状况。
- T 　显示现行终端机下的所有程序。
- u 　以用户为主的格式来显示程序状况。
- U<帐号名称> 　列出属于该用户的程序的状况。
- v 　采用虚拟内存的格式显示程序状况。
- w 　采用宽阔的格式来显示程序状况。
- x 　显示所有程序，不以终端机来区分。
- X 　采用旧试的Linux i386登陆格式显示程序状况。

实例

显示指定用户信息

```
# gitps hnlinux
```

Linux uname命令

Linux uname命令用于显示系统信息。

uname可显示电脑以及操作系统的相关信息。

语法

```
uname [-amnrsv][--help][--version]
```

参数说明：

- -a或--all 　显示全部的信息。
- -m或--machine 　显示电脑类型。
- -n或-nodename 　显示在网络上的主机名称。
- -r或--release 　显示操作系统的发行编号。
- -s或--sysname 　显示操作系统名称。
- -v 　显示操作系统的版本。
- --help 　显示帮助。
- --version 　显示版本信息。

实例

显示系统信息

```
# uname -a
Linux snail-hnlinux 2.6.32-21-generic #32-Ubuntu SMP Fri Apr 16 08:10:02 UTC 2010 i686 GN
```

显示计算机类型

```
# uname -m
i686
```

显示计算机名

```
# uname -n
snail-hnlinux
```

显示操作系统发行编号

```
# uname -r
2.6.32-21-generic
```

显示操作系统名称

```
# uname -s
Linux
```

显示系统时间

```
# uname -v
#32-Ubuntu SMP Fri Apr 16 08:10:02 UTC 2014
```

Linux logrotate命令

Linux logrotate命令用于管理记录文件。

使用logrotate指令，可让你轻松管理系统所产生的记录文件。它提供自动替换，压缩，删除和邮寄记录文件，每个记录文件都可被设置成每日，每周或每月处理，也能在文件太大时立即处理。您必须自行编辑，指定配置文件，预设的配置文件存放在/etc目录下，文件名称为logrotate.conf。

语法

```
logrotate [-?dfv][-s <状态文件>][--usage][配置文件]
```

参数说明：

- -?或--help　在线帮助。
- -d或--debug　详细显示指令执行过程，便于排错或了解程序执行的情况。
- -f或--force　强行启动记录文件维护操作，纵使logrotate指令认为没有需要亦然。
- -s<状态文件>或--state=<状态文件>　使用指定的状态文件。
- -v或--version　显示指令执行过程。
- -usage　显示指令基本用法。

实例

指定记录文件

```
# logrotate /root/log.config
```

Linux tload命令

Linux tload命令用于显示系统负载状况。

tload指令使用ASCII字符简单地以文字模式显示系统负载状态。假设不给予终端机编号，则会在执行tload指令的终端机显示负载情形。

语法

```
tload [-V][-d <间隔秒数>][-s <刻度大小>][终端机编号]
```

参数说明：

- -d<间隔秒数> 设置tload检测系统负载的间隔时间，单位以秒计算。
- -s<刻度大小> 设置图表的垂直刻度大小，单位以列计算。
- -V 显示版本信息。

实例

显示系统负载

```
# tload
```

Linux swatch命令

Linux swatch命令用于系统监控程序。

swatch可用来监控系统记录文件，并在发现特定的事件时，执行指定的动作。swatch所监控的事件以及对应事件的动作都存放在swatch的配置文件中。预设的配置文件为拥护根目录下的.swatchrc。然而在Red Hat Linux的预设用户根目录下并没有.swatchrc配置文件，您可将/usr/doc/swatch-2.2/config_files/swatchrc.personal文件复制到用户根目录下的.swatchrc，然后修改.swatchrc所要监控的事件及执行的动作。

语法

```
swatch [-A <分隔字符>][-c <设置文件>][-f <记录文件>][-I <分隔字符>][-P <分隔字符>][-r <时间>][-t
```

参数说明：

- -A<分隔字符> 预设配置文件中，动作的分隔字符，预设为逗号。
- -c<设置文件> 指定配置文件，而不使用预设的配置文件。
- -f<记录文件> 检查指定的记录文件，检查完毕后不会继续监控该记录文件。
- -I<分隔字符> 指定输入记录的分隔字符，预设为换行字符。
- -P<分隔字符> 指定配置文件中，事件的分隔字符，预设为逗号。
- -r<时间> 在指定的时间重新启动。
- -t<记录文件> 检查指定的记录文件，并且会监控加入记录文件中的后继记录。

实例

开启系统监视

```
# swatch
```

Linux chsh命令

Linux chsh命令用于更改使用者 shell 设定。

使用权限：所有使用者。

语法

```
shell>> chsh
```

实例

```
shell>> chsh
Changing fihanging shell for user1
Password: [del]
New shell [/bin/tcsh]: ### [是目前使用的 shell]
[del]
shell>> chsh -l ### 展示 /etc/shells 档案内容
/bin/bash
/bin/sh
/bin/ash
/bin/bsh
/bin/tcsh
/bin/csh
```

改变当前的shell。当前的shell 设置为//bin/bash，通过chsh命令，改变shell的设置/bin/csh。

```
# chsh
Changing shell for root.
New shell [/bin/bash]: /bin/csh //输入新的shell地址
Shell changed.
```

通过 -s 参数改变当前的shell设置

```
# chsh -s /bin/csh //改变当前设置为 /bin/csh
Changing shell for root.
Shell not changed.
```

Linux whoami命令

Linux whoami命令用于显示自身用户名称。

显示自身的用户名称，本指令相当于执行"id -un"指令。

语法

```
whoami [--help][--version]
```

参数说明：

- --help 　在线帮助。
- --version 　显示版本信息。

实例

显示用户名

```
# whoami
root
```

Linux who命令

Linux who命令用于显示系统中有哪些使用者正在上面，显示的资料包含了使用者 ID、使用的终端机、从哪边连上来的、上线时间、呆滞时间、CPU 使用量、动作等等。

使用权限：所有使用者都可使用。

语法

```
who - [husfV] [user]
```

参数说明：

- -h : 不要显示标题列
- -u : 不要显示使用者的动作/工作
- -s : 使用简短的格式来显示
- -f : 不要显示使用者的上线位置
- -V : 显示程序版本

实例

显示当前登录系统的用户

```
# who  //显示当前登录系统的用户
root   tty7     2014-05-13 12:12 (:0)
root   pts/0    2014-05-14 17:09 (:0.0)
root   pts/1    2014-05-14 18:51 (192.168.1.17)
root   pts/2    2014-05-14 19:48 (192.168.1.17)
```

显示标题栏

```
# who -H
NAME   LINE     TIME        COMMENT
root   tty7     2014-05-13 12:12 (:0)
root   pts/0    2014-05-14 17:09 (:0.0)
root   pts/1    2014-05-14 18:51 (192.168.1.17)
root   pts/2    2014-05-14 19:48 (192.168.1.17)
```

显示用户登录来源

```
# who -l -H
NAME   LINE     TIME        IDLE     PID COMMENT
LOGIN  tty4     2014-05-13 12:11         852 id=4
LOGIN  tty5     2014-05-13 12:11         855 id=5
LOGIN  tty2     2014-05-13 12:11         862 id=2
LOGIN  tty3     2014-05-13 12:11         864 id=3
LOGIN  tty6     2014-05-13 12:11         867 id=6
LOGIN  tty1     2014-05-13 12:11        1021 id=1
```

显示终端属性

```
# who -T -H
NAME    LINE     TIME        COMMENT
root   + tty7     2014-05-13 12:12 (:0)
root   + pts/0    2014-05-14 17:09 (:0.0)
root   - pts/1    2014-05-14 18:51 (192.168.1.17)
root   - pts/2    2014-05-14 19:48 (192.168.1.17)
```

只显示当前用户

```
# who -m -H
NAME   LINE     TIME        COMMENT
root   pts/1    2014-05-14 18:51 (192.168.1.17)
```

精简模式显示

```
# who -q
root root root root
# users=4
```

Linux vlock命令

Linux vlock命令用于锁住虚拟终端。

执行vlock(virtual console lock)指令可锁住虚拟终端，避免他人使用。

语法

```
vlock [-achv]
```

参数说明：

- -a或--all 锁住所有的终端阶段作业，如果您在全屏幕的终端中使用本参数，则会将用键盘
- 切换终端机的功能一并关闭。
- -c或--current 锁住目前的终端阶段作业，此为预设值。
- -h或--help 在线帮助。
- -v或--version 显示版本信息。

实例

锁定虚拟终端

```
# vlock
```

Linux usermod命令

Linux usermod命令用于修改用户帐号。

usermod可用来修改用户帐号的各项设定。

语法

```
usermod [-LU][-c <备注>][-d <登入目录>][-e <有效期限>][-f <缓冲天数>][-g <群组>][-G <群组>][-l
```

参数说明：

- -c<备注>　修改用户帐号的备注文字。
- -d登入目录>　修改用户登入时的目录。
- -e<有效期限>　修改帐号的有效期限。
- -f<缓冲天数>　修改在密码过期后多少天即关闭该帐号。
- -g<群组>　修改用户所属的群组。
- -G<群组>　修改用户所属的附加群组。
- -l<帐号名称>　修改用户帐号名称。
- -L　锁定用户密码，使密码无效。
- -s<shell>　修改用户登入后所使用的shell。
- -u<uid>　修改用户ID。
- -U　解除密码锁定。

实例

更改登录目录

```
# usermod -d /home/hnlinux root
```

改变用户的uid

```
# usermod -u 777 root
```

Linux userdel命令

Linux userdel命令用于删除用户帐号。

userdel可删除用户帐号与相关的文件。若不加参数，则仅删除用户帐号，而不删除相关文件。

语法

```
userdel [-r][用户帐号]
```

参数说明：

- -r　删除用户登入目录以及目录中所有文件。

实例

删除用户账号

```
# userdel hnlinux
```

Linux userconf命令

Linux userconf命令用于用户帐号设置程序。

userconf实际上为linuxconf的符号连接，提供图形界面的操作方式，供管理员建立与管理各类帐号。若不加任何参数，即进入图形界面。

语法

```
userconf [--addgroup <群组>][--adduser <用户ID><群组><用户名称><shell>][--delgroup <群组>][--
```

参数说明：

- --addgroup<群组> 新增群组。
- --adduser<用户ID><群组><用户名称><shell> 新增用户帐号。
- --delgroup<群组> 删除群组。
- --deluser<用户ID> 删除用户帐号。
- --help 显示帮助。

实例

新增用户

```
# userconf --adduser 666 tt lord /bin/bash //新增用户账号
```

Linux id命令

Linux id命令用于显示用户的ID，以及所属群组的ID。

id会显示用户以及所属群组的实际与有效ID。若两个ID相同，则仅显示实际ID。若仅指定用户名称，则显示目前用户的ID。

语法

```
id [-gGnru][--help][--version][用户名称]
```

参数说明：

- -g或--group　显示用户所属群组的ID。
- -G或--groups　显示用户所属附加群组的ID。
- -n或--name　显示用户，所属群组或附加群组的名称。
- -r或--real　显示实际ID。
- -u或--user　显示用户ID。
- -help　显示帮助。
- -version　显示版本信息。

实例

显示当前用户信息

```
# id //显示当前用户ID
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel) c
```

显示用户群组的ID

```
# id -g
0
```

显示所有群组的ID

```
# id -g
0 1 2 3 4 5 6 10
```

显示指定用户信息

```
# id hnlinux
```

Linux w命令

Linux w命令用于显示目前登入系统的用户信息。

执行这项指令可得知目前登入系统的用户有哪些人，以及他们正在执行的程序。

单独执行 w 指令会显示所有的用户，您也可指定用户名称，仅显示某位用户的相关信息。

语法

```
w [-fhlsuV][用户名称]
```

参数说明：

- -f 　开启或关闭显示用户从何处登入系统。
- -h 　不显示各栏位的标题信息列。
- -l 　使用详细格式列表，此为预设值。
- -s 　使用简洁格式列表，不显示用户登入时间，终端机阶段作业和程序所耗费的CPU时间。
- -u 　忽略执行程序的名称，以及该程序耗费CPU时间的信息。
- -V 　显示版本信息。

实例

显示当前用户

```
w  //显示当前用户，不显示登录位置
19:50:14 up 9:27, 4 users, load average: 0.31, 0.26, 0.18
USER   TTY   FROM       LOGIN@  IDLE  JCPU  PCPU WHAT
root   tty7   :0        Thu12  31:39m 10:10  0.60s gnome-session
root   pts/0  :0.0       17:09  2:18m 15.26s 0.15s bash
root   pts/1  192.168.1.17   18:51  1.00s 1.24s 0.14s -bash
root   pts/2  192.168.1.17   19:48  60.00s 0.05s 0.05s -bash
```

不显示登录位置

```
w -f
19:53:59 up 9:31, 4 users, load average: 0.05, 0.16, 0.15
USER   TTY    LOGIN@  IDLE  JCPU  PCPU WHAT
root   tty7   Thu12  31:43m 10:10  0.60s gnome-session
root   pts/0   17:09  2:21m 15.26s 0.15s bash
root   pts/1   18:51  0.00s 1.04s 0.14s -bash
root   pts/2   19:48  4:45  0.05s 0.05s -bash
```

以精简模式显示

```
w -s
19:54:37 up 9:31, 4 users, load average: 0.24, 0.19, 0.16
USER   TTY   FROM        IDLE WHAT
root   tty7   :0        31:43m gnome-session
root   pts/0  :0.0       2:22m bash
root   pts/1  192.168.1.17   0.00s -bash
root   pts/2  192.168.1.17   5:23 -bash
```

不显示标题

```
w -h
root   tty7   :0        Thu12  31:44m 10:10  0.60s gnome-session
root   pts/0  :0.0       17:09  2:23m 15.26s 0.15s bash
root   pts/1  192.168.1.17   18:51  0.00s 1.05s 0.14s -bash
root   pts/2  192.168.1.17   19:48  5:54  0.05s 0.05s -bash
```

Linux skill命令

Linux skill命令送个讯号给正在执行的程序，预设的讯息为 TERM (中断)，较常使用的讯息为 HUP、INT、KILL、STOP、CONT 和 0。

讯息有三种写法：分别为 -9、-SIGKILL、-KILL，可以使用 -l 或 -L 已列出可使用的讯息。

使用权限：所有使用者。

其他相关的命令：kill

语法

```
skill [signal to send] [options] 选择程序的规则
```

一般参数：

- -f 快速模式/尚未完成
- -i 互动模式/ 每个动作将要被确认
- -v 详细输出/ 列出所选择程序的资讯
- -w 智能警告讯息/ 尚未完成
- -n 没有动作/ 显示程序代号

参数：选择程序的规则可以是：终端机代号、使用者名称、程序代号、命令名称。

- -t 终端机代号 (tty 或 pty)
- -u 使用者名称
- -p 程序代号 (pid)
- -c 命令名称可使用的讯号

以下列出已知的讯号名称、讯号代号、功能。

名称（代号）	功能/描述
ALRM 14	离开
HUP 1	离开
INT 2	离开
KILL 9	离开/强迫关闭
PIPE 13	离开
POLL	离开
PROF	离开

TERM 15	离开
USR1	离开
USR2	离开
VTALRM	离开
STKFLT	离开/只适用于i386、m68k、arm 和 ppc 硬件
UNUSED	离开/只适用于i386、m68k、arm 和 ppc 硬件
TSTP	停止/产生与内容相关的行为
TTIN	停止/产生与内容相关的行为
TTOU	停止/产生与内容相关的行为
STOP	停止/强迫关闭
CONT	重新启动/如果在停止状态则重新启动，否则忽略
PWR	忽略/在某些系统中会离开
WINCH	忽略
CHLD	忽略
ABRT 6	核心
FPE 8	核心
ILL 4	核心
QUIT 3	核心
SEGV 11	核心
TRAP 5	核心
SYS	核心/或许尚未实作
EMT	核心/或许尚未实作
BUS	核心/核心失败
XCPU	核心/核心失败
XFSZ	核心/核心失败

实例

停止所有在 PTY 装置上的程序

```
skill -KILL -v pts/*
```

停止三个使用者 user1、user2、user3

```
skill -STOP user1 user2 user3
```

Linux su命令

Linux su命令用于变更为其他使用者的身份，除 root 外，需要键入该使用者的密码。

使用权限：所有使用者。

语法

```
su [-fmp] [-c command] [-s shell] [--help] [--version] [-] [USER [ARG]]
```

参数说明：

- -f 或 --fast 不必读启动档（如 csh.cshrc 等），仅用于 csh 或 tcsh
- -m -p 或 --preserve-environment 执行 su 时不改变环境变数
- -c command 或 --command=command 变更为帐号为 USER 的使用者并执行指令（command）后再变回原来使用者
- -s shell 或 --shell=shell 指定要执行的 shell （bash csh tcsh 等），预设值为 /etc/passwd 内的该使用者（USER） shell
- --help 显示说明文件
- --version 显示版本资讯
-
    - -l 或 --login 这个参数加了之后，就好像是重新 login 为该使用者一样，大部份环境变数（HOME SHELL USER等等）都是以该使用者（USER）为主，并且工作目录也会改变，如果没有指定 USER ，内定是 root
- USER 欲变更的使用者帐号
- ARG 传入新的 shell 参数

实例

变更帐号为 root 并在执行 ls 指令后退出变回原使用者

```
su -c ls root
```

变更帐号为 root 并传入 -f 参数给新执行的 shell

```
su root -f
```

变更帐号为 clsung 并改变工作目录至 clsung 的家目录（home dir）

```
su - clsung
```

切换用户

```
hnlinux@w3cschool.cc:~$ whoami //显示当前用户
hnlinux
hnlinux@w3cschool.cc:~$ pwd //显示当前目录
/home/hnlinux
hnlinux@w3cschool.cc:~$ su root //切换到root用户
密码：
root@w3cschool.cc:/home/hnlinux# whoami
root
root@w3cschool.cc:/home/hnlinux# pwd
/home/hnlinux
```

切换用户，改变环境变量

```
hnlinux@w3cschool.cc:~$ whoami //显示当前用户
hnlinux
hnlinux@w3cschool.cc:~$ pwd //显示当前目录
/home/hnlinux
hnlinux@w3cschool.cc:~$ su - root //切换到root用户
密码：
root@w3cschool.cc:/home/hnlinux# whoami
root
root@w3cschool.cc:/home/hnlinux# pwd //显示当前目录
/root
```

Linux renice命令

Linux renice命令用于重新指定一个或多个行程（Process）的优先序（一个或多个将根据参数而定）。

注意：每一个行程（Process）都有一个唯一的（unique）id。

使用权限：所有使用者。

语法

```
renice priority [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]
```

参数说明：

- -p pid 重新指定行程的 id 为 pid 的行程的优先序
- -g pgrp 重新指定行程群组(process group)的 id 为 pgrp 的行程 (一个或多个) 的优先序
- -u user 重新指定行程拥有者为 user 的行程的优先序

实例

将行程 id 为 987 及 32 的行程与行程拥有者为 daemon 及 root 的优先序号码加 1

```
renice +1 987 -u daemon root -p 32
```

Linux newgrp命令

Linux newgrp命令用于登入另一个群组。

newgrp指令类似login指令，当它是以相同的帐号，另一个群组名称，再次登入系统。欲使用newgrp指令切换群组，您必须是该群组的用户，否则将无法登入指定的群组。单一用户要同时隶属多个群组，需利用交替用户的设置。若不指定群组名称，则newgrp指令会登入该用户名称的预设群组。

语法

```
newgrp [群组名称]
```

实例

改变群组

```
# newgrp root
```

Linux whois命令

Linux whois命令用于查找并显示用户信息。

whois指令会去查找并显示指定帐号的用户相关信息，因为它是到Network Solutions的WHOIS数据库去查找，所以该帐号名称必须在上面注册方能寻获，且名称没有大小写的差别。

语法

```
whois [帐号名称]
```

实例

显示指定用户信息

```
# whois root

//查找root用户信息
```

查询域名描述信息

```
# whois .Lx138.COm

Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

...省略部分内容
```

查询域名信息

```
# whois Lx138.COm

The Registry database contains ONLY .COM, .NET, .EDU domains and
Registrars.
Domain Name ..................... Lx138.COm
Name Server ..................... dns15.hichina.com
                 dns16.hichina.com
Registrant ID ................... hc937242545-cn

...省略部分内容
```

查询域名信息省略法律声明

```
# whois -H Lx138.COm
...省略内容
```

指定端口查询

```
# whois -p 80 Lx138.COm
...省略内容
```

Linux free命令

Linux free命令用于显示内存状态。

free指令会显示内存的使用情况，包括实体内存，虚拟的交换文件内存，共享内存区段，以及系统核心使用的缓冲区等。

语法

```
free [-bkmotV][-s <间隔秒数>]
```

参数说明：

- -b 　以Byte为单位显示内存使用情况。
- -k 　以KB为单位显示内存使用情况。
- -m 　以MB为单位显示内存使用情况。
- -o 　不显示缓冲区调节列。
- -s<间隔秒数> 　持续观察内存使用状况。
- -t 　显示内存总和列。
- -V 　显示版本信息。

实例

显示内存使用情况

```
# free //显示内存使用信息
total used free shared buffers cached
Mem: 254772 184568 70204 0 5692 89892
-/+ buffers/cache: 88984 165788
Swap: 524280 65116 459164
```

以总和的形式显示内存的使用信息

```
# free -t //以总和的形式查询内存的使用信息
total used free shared buffers cached
Mem: 254772 184868 69904 0 5936 89908
-/+ buffers/cache: 89024 165748
Swap: 524280 65116 459164
Total: 779052 249984 529068
```

周期性的查询内存使用信息

```
# free -s 10 //每10s 执行一次命令
total used free shared buffers cached
Mem: 254772 187628 67144 0 6140 89964
-/+ buffers/cache: 91524 163248
Swap: 524280 65116 459164

total used free shared buffers cached
Mem: 254772 187748 67024 0 6164 89940
-/+ buffers/cache: 91644 163128
Swap: 524280 65116 459164
```

Linux命令大全 - 系统设定

reset	clear	alias	dircolors
aumix	bind	chroot	clock
crontab	declare	depmod	dmesg
enable	eval	export	pwunconv
grpconv	rpm	insmod	kbdconfig
lilo	liloconfig	lsmod	minfo
set	modprobe	ntsysv	mouseconfig
passwd	pwconv	rdate	resize
rmmod	grpunconv	modinfo	time
setup	sndconfig	setenv	setconsole
timeconfig	ulimit	unset	chkconfig
apmd	hwclock	mkkickstart	fbset
unalias	SVGATextMode		

Linux bind命令

Linux bind命令用于显示或设置键盘按键与其相关的功能。

您可以利用bind命令了解有哪些按键组合与其功能，也可以自行指定要用哪些按键组合。

语法

```
bind [-dlv][-f <按键配置文件>][-m <按键配置>][-q <功能>]
```

参数说明：

- -d 　显示按键配置的内容。
- -f<按键配置文件> 　载入指定的按键配置文件。
- -l 　列出所有的功能。
- -m<按键配置> 　指定按键配置。
- -q<功能> 　显示指定功能的按键。
- -v 　列出目前的按键配置与其功能。

实例

显示按键组合的所有功能

```
# bind -l //显示按键组合的内容
abort
accept-line
alias-expand-line
arrow-key-prefix
backward-byte
backward-char
backward-delete-char
backward-kill-line
backward-kill-word
backward-word
beginning-of-history
beginning-of-line
……省略部分内容
vi-goto-mark
vi-insert-beg
vi-insertion-mode
vi-match
vi-movement-mode
vi-next-word
vi-overstrike
vi-overstrike-delete
vi-prev-word
vi-put
vi-redo
vi-replace
vi-rubout
vi-search
vi-search-again
vi-set-mark
vi-subst
vi-tilde-expand
vi-yank-arg
vi-yank-to
yank
yank-last-arg
yank-nth-arg
yank-pop
```

显示当前按键组合的设置

```
# bind -l
abort
accept-line
alias-expand-line
arrow-key-prefix
backward-byte
backward-char
backward-delete-char
backward-kill-line
backward-kill-word
backward-word
beginning-of-history
beginning-of-line
call-last-kbd-macro
capitalize-word
character-search
character-search-backward
clear-screen
complete
complete-command
complete-filename
complete-hostname
complete-into-braces
complete-username
complete-variable
copy-backward-word
copy-forward-word
```

```
copy-region-as-kill
dabbrev-expand
delete-char
delete-char-or-list
delete-horizontal-space
digit-argument
display-shell-version
do-lowercase-version
downcase-word
dump-functions
dump-macros
dump-variables
dynamic-complete-history
edit-and-execute-command
emacs-editing-mode
end-kbd-macro
end-of-history
end-of-line
exchange-point-and-mark
forward-backward-delete-char
forward-byte
forward-char
forward-search-history
forward-word
glob-complete-word
glob-expand-word
glob-list-expansions
history-and-alias-expand-line
history-expand-line
history-search-backward
history-search-forward
insert-comment
insert-completions
insert-last-argument
kill-line
kill-region
kill-whole-line
kill-word
magic-space
menu-complete
menu-complete-backward
next-history
non-incremental-forward-search-history
non-incremental-forward-search-history-again
non-incremental-reverse-search-history
non-incremental-reverse-search-history-again
old-menu-complete
operate-and-get-next
overwrite-mode
possible-command-completions
possible-completions
possible-filename-completions
possible-hostname-completions
possible-username-completions
possible-variable-completions
previous-history
quoted-insert
redraw-current-line
re-read-init-file
reverse-search-history
revert-line
self-insert
set-mark
shell-backward-kill-word
shell-backward-word
shell-expand-line
shell-forward-word
shell-kill-word
skip-csi-sequence
start-kbd-macro
tab-insert
tilde-expand
```

```
transpose-chars
transpose-words
tty-status
undo
universal-argument
unix-filename-rubout
unix-line-discard
unix-word-rubout
upcase-word
vi-append-eol
vi-append-mode
vi-arg-digit
vi-back-to-indent
vi-bword
vi-bWord
vi-change-case
vi-change-char
vi-change-to
vi-char-search
vi-column
vi-complete
vi-delete
vi-delete-to
vi-editing-mode
vi-end-word
vi-eof-maybe
vi-eword
vi-eWord
vi-fetch-history
vi-first-print
vi-fword
vi-fWord
vi-goto-mark
vi-insert-beg
vi-insertion-mode
vi-match
vi-movement-mode
vi-next-word
vi-overstrike
vi-overstrike-delete
vi-prev-word
vi-put
vi-redo
vi-replace
vi-rubout
vi-search
vi-search-again
vi-set-mark
vi-subst
vi-tilde-expand
vi-yank-arg
vi-yank-to
yank
yank-last-arg
yank-nth-arg
yank-pop
root@snail-hnlinux:~#
root@snail-hnlinux:~#
root@snail-hnlinux:~#
root@snail-hnlinux:~#
root@snail-hnlinux:~# bind -v
set bind-tty-special-chars on
set blink-matching-paren on
set byte-oriented off
set completion-ignore-case off
set convert-meta off
set disable-completion off
set echo-control-characters on
set enable-keypad off
set enable-meta-key on
set expand-tilde off
set history-preserve-point off
```

```
set horizontal-scroll-mode off
set input-meta on
set mark-directories on
set mark-modified-lines off
set mark-symlinked-directories off
set match-hidden-files on
set meta-flag on
set output-meta on
set page-completions on
set prefer-visible-bell on
set print-completions-horizontally off
set revert-all-at-newline off
set show-all-if-ambiguous off
set show-all-if-unmodified off
set skip-completed-text off
set visible-stats off
set bell-style audible
set comment-begin #
set completion-prefix-display-length 0
set completion-query-items 100
set editing-mode emacs
set history-size 1000
set keymap emacs
```

列出指定功能的按键和按键组合

```
# bind -q abort
//请用 调用abort “C-g”, “C-xC-g”, “eC-g”.

# bind -q accept-line //列出功能“accept-line”按键以及组合按键
//请用 调用accept-line “C-j”, “C-m”.
```

Linux aumix命令

Linux aumix命令用于设置音效装置。

aumix(audio mixer)命令设置各项音效装置的信号强度以及指定播放与录音的装置。

语法

```
aumix [-123bcilmoprstvwWx][(+/-)强度][PqR][-dfhILqS]
```

参数说明：[-123bcilmoprstvwWx]为频道参数，用来指定装置的频道；[PqR]可用来指定播放或录音装置；[-dfhILqS] 则为指令参数。若不加任何参数，aumix会显示简单的图形界面供调整设置频道参数。

- -1 输入信号线 1 。
- -2 输入信号线 2 。
- -3 输入信号线 3 。
- -b 低音。
- -c CD。
- -i 输入信号强度。
- -m 麦克风。
- -o 输出信号强度。
- -p PC喇叭。
- -r 录音。
- -s 合成器。
- -t 高音。
- -v 主音量。
- -w PCM。
- -W PCM2。
- -x 混音器。
- (+/-)强度 出现(+/-)时，代表在原有的强度上加减指定值。若未使用(+/-)，则直接将强度设为指定值。 指定音效装置
- P 指定播放装置。
- q 显示频道设置。
- R 指定录音装置。

指令参数：

- -d 指定音效装置的名称。
- -f 指定存储或载入设置的文件。

- -h　在使用时显示信息。
- -I　以图形界面方式来执行aumix。
- -L　从$HOME/.aumixrc或/etc/aumixrc载入设置。
- -q　显示所有频道的设置值。
- -S　将设置值保存至/HOME/.aumixrc。

实例

设置音效设备

```
# aumix
```

Linux dircolors命令

Linux dircolors命令用于设置 ls 指令在显示目录或文件时所用的色彩。

dircolors可根据[色彩配置文件]来设置LS_COLORS环境变量或是显示设置LS_COLORS环境变量的shell指令。

语法

```
dircolors [色彩配置文件]
```

或

```
dircolors [-bcp][--help][--version]
```

参数说明：

- -b或--sh或--bourne-shell 　显示在Boume shell中，将LS_COLORS设为目前预设置的shell指令。
- -c或--csh或--c-shell 　显示在C shell中，将LS_COLORS设为目前预设置的shell指令。
- -p或--print-database 　显示预设置
- -help 　显示帮助。
- -version 　显示版本信息。

实例

显示默认值

```
# dircolors -p //显示默认值
# Configuration file for dircolors, a utility to help you set the
# LS_COLORS environment variable used by GNU ls with the --color option.
# Copyright (C) 1996, 1999-2008
# Free Software Foundation, Inc.
# Copying and distribution of this file, with or without modification,
# are permitted provided the copyright notice and this notice are preserved.
# The keywords COLOR, OPTIONS, and EIGHTBIT (honored by the
# slackware version of dircolors) are recognized but ignored.
# Below, there should be one TERM entry for each termtype that is colorizable
TERM Eterm
TERM ansi
TERM color-xterm
TERM con132x25
TERM con132x30
TERM con132x43
TERM con132x60
TERM con80x25
TERM con80x28
TERM xterm-debian
```

```
# Below are the color init strings for the basic file types. A color init
# string consists of one or more of the following numeric codes:
# Attribute codes:
# 00=none 01=bold 04=underscore 05=blink 07=reverse 08=concealed
# Text color codes:
# 30=black 31=red 32=green 33=yellow 34=blue 35=magenta 36=cyan 37=white
# Background color codes:
# 40=black 41=red 42=green 43=yellow 44=blue 45=magenta 46=cyan 47=white
#NORMAL 00 # no color code at all
#FILE 00 # regular file: use no color at all
RESET 0 # reset to “normal“ color
DIR 01;34 # directory
LINK 01;36 # symbolic link. (If you set this to 'target' instead of a
# numerical value, the color is as for the file pointed to.)
HARDLINK 44;37 # regular file with more than one link
FIFO 40;33 # pipe
SOCK 01;35 # socket
DOOR 01;35 # door
BLK 40;33;01 # block device driver
CHR 40;33;01 # character device driver
ORPHAN 40;31;01 # symlink to nonexistent file, or non-stat'able file
SETUID 37;41 # file that is setuid (u+s)
SETGID 30;43 # file that is setgid (g+s)
CAPABILITY 30;41 # file with capability
STICKY_OTHER_WRITABLE 30;42 # dir that is sticky and other-writable (+t,o+w)
OTHER_WRITABLE 34;42 # dir that is other-writable (o+w) and not sticky
STICKY 37;44 # dir with the sticky bit set (+t) and not other-writable
# This is for files with execute permission:
EXEC 01;32
# List any file extensions like '.gz' or '.tar' that you would like ls
# to colorize below. Put the extension, a space, and the color init string.
# (and any comments you want to add after a '#')
# If you use DOS-style suffixes, you may want to uncomment the following:
#.cmd 01;32 # executables (bright green)
#.exe 01;32
#.com 01;32
#.btm 01;32
#.bat 01;32
# Or if you want to colorize scripts even if they do not have the
# executable bit actually set.
#.sh 01;32
#.csh 01;32
# archives or compressed (bright red)
.tar 01;31

.pcx 01;35
.mov 01;35
.mpg 01;35
.mpeg 01;35
.m2v 01;35
.mkv 01;35
.ogm 01;35
.mp4 01;35
.m4v 01;35
.mp4v 01;35
.vob 01;35
.qt 01;35
.nuv 01;35
.wmv 01;35
.asf 01;35
.rm 01;35
.rmvb 01;35
.flc 01;35
.avi 01;35
.fli 01;35
.flv 01;35
.gl 01;35
.dl 01;35
.xcf 01;35
.xwd 01;35
.yuv 01;35
# http://wiki.xiph.org/index.php/MIME_Types_and_File_Extensions
```

```
.axv 01;35
.anx 01;35
.ogv 01;35
.ogx 01;35
# audio formats
.aac 00;36
.au 00;36
.flac 00;36
.mid 00;36
.midi 00;36
.mka 00;36
.mp3 00;36
.mpc 00;36
.ogg 00;36
.ra 00;36
.wav 00;36
# http://wiki.xiph.org/index.php/MIME_Types_and_File_Extensions
.axa 00;36
.oga 00;36
.spx 00;36
.xspf 00;36
```

Linux alias命令

Linux alias命令用于设置指令的别名。

用户可利用alias，自定指令的别名。若仅输入alias，则可列出目前所有的别名设置。alias的效力仅及于该次登入的操作。若要每次登入是即自动设好别名，可在.profile或.cshrc中设定指令的别名。

语法

```
alias[别名]=[指令名称]
```

参数说明：若不加任何参数，则列出目前所有的别名设置。

实例

给命令设置别名

```
# alias lx=ls
# lx
anaconda-ks.cfg Desktop install.log install.log.syslog qte
```

Linux clear命令

Linux clear命令用于清除屏幕。

语法

```
clear
```

实例

清屏

```
#clear
```

Linux reset命令

Linux reset命令其实和 tset 是一同个命令，它的用途是设定终端机的状态。一般而言，这个命令会自动的从环境变数、命令列或是其它的组态档决定目前终端机的型态。如果指定型态是 '?' 的话，这个程序会要求使用者输入终端机的型别。

由于这个程序会将终端机设回原始的状态，除了在 login 时使用外，当系统终端机因为程序不正常执行而进入一些奇怪的状态时，你也可以用它来重设终端机o 例如不小心把二进位档用 cat 指令进到终端机，常会有终端机不再回应键盘输入，或是回应一些奇怪字元的问题。此时就可以用 reset 将终端机回复至原始状态。

语法

```
tset [-IQqrs] [-] [-e ch] [-i ch] [-k ch] [-m mapping] [terminal]
```

参数说明：

- -p 将终端机类别显示在屏幕上，但不做设定的动作。这个命令可以用来取得目前终端机的类别。
- -e ch 将 erase 字元设成 ch
- -i ch 将中断字元设成 ch
- -k ch 将删除一行的字元设成 ch
- -I 不要做设定的动作，如果没有使用选项 -Q 的话，erase、中断及删除字元的目前值依然会送到屏幕上。
- -Q 不要显示 erase、中断及删除字元的值到屏幕上。
- -r 将终端机类别印在屏幕上。
- -s 将设定 TERM 用的命令用字串的型式送到终端机中，通常在 .login 或 .profile 中用。

实例

让使用者输入一个终端机型别并将终端机设到该型别的预设状态

```
# reset ?
```

将 erase 字元设定 control-h

```
# reset -e ^B
```

将设定用的字串显示在屏幕上

```
# reset -s
Erase is control-B (^B).
Kill is control-U (^U).
Interrupt is control-C (^C).
TERM=xterm;
```

Linux enable命令

Linux enable命令用于启动或关闭 shell 内建指令。

若要执行的文件名称与shell内建指令相同，可用enable -n来关闭shell内建指令。若不加-n参数，enable可重新启动关闭的指令。

语法

```
enable [-n][-all][内建指令]
```

参数说明：

- -n 关闭指定的shell内建指令。
- -all 显示shell所有关闭与启动的指令。

实例

显示shell内置命令

```
# enable //显示shell命令
enable .
enable :
enable [
enable alias
enable bg
enable bind
enable break
enable builtin
enable caller
enable cd
enable command
enable compgen
enable complete
enable compopt
enable continue
enable declare
enable dirs
enable disown
enable echo
enable enable
enable eval
enable exec
enable exit
enable export
enable false
enable fc
enable fg
enable getopts
enable hash
enable help
enable history
enable jobs
enable kill
enable let
enable local
enable logout
enable mapfile
enable popd
enable printf
enable pushd
enable pwd
enable read
enable readarray
enable readonly
enable return
enable set
enable shift
enable shopt
enable source
enable suspend
enable test
enable times
enable trap
enable true
enable type
enable typeset
enable ulimit
enable umask
enable unalias
enable unset
enable wait
```

Linux dmesg命令

Linux dmesg命令用于显示开机信息。

kernel会将开机信息存储在ring buffer中。您若是开机时来不及查看信息，可利用dmesg来查看。开机信息亦保存在/var/log目录中，名称为dmesg的文件里。

语法

```
dmesg [-cn][-s <缓冲区大小>]
```

参数说明：

- -c 显示信息后，清除ring buffer中的内容。
- -s<缓冲区大小> 预设置为8196，刚好等于ring buffer的大小。
- -n 设置记录信息的层级。

实例

显示开机信息

```
# dmesg |less
WARNING: terminal is not fully functional
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 2.6.32-21-generic (buildd@rothera) (gcc version 4.4.3 (Ub
untu 4.4.3-4ubuntu5) ) #32-Ubuntu SMP Fri Apr 16 08:10:02 UTC 2010 (Ubuntu 2.6.32-21.3
2-generic 2.6.32.11+drm33.2)
[    0.000000] KERNEL supported cpus:
[    0.000000]  Intel GenuineIntel
[    0.000000]  AMD AuthenticAMD
[    0.000000]  NSC Geode by NSC
[    0.000000]  Cyrix CyrixInstead
[    0.000000]  Centaur CentaurHauls
[    0.000000]  Transmeta GenuineTMx86
[    0.000000]  Transmeta TransmetaCPU
[    0.000000]  UMC UMC UMC UMC
[    0.000000] BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: 0000000000000000 - 000000000009f800 (usable)
[    0.000000] BIOS-e820: 000000000009f800 - 00000000000a0000 (reserved)
[    0.000000] BIOS-e820: 00000000000ca000 - 00000000000cc000 (reserved)
[    0.000000] BIOS-e820: 00000000000dc000 - 00000000000e0000 (reserved)
[    0.000000] BIOS-e820: 00000000000e4000 - 0000000000100000 (reserved)
[    0.000000] BIOS-e820: 0000000000100000 - 000000003fef0000 (usable)
[    0.000000] BIOS-e820: 000000003fef0000 - 000000003feff000 (ACPI data)
[    0.000000] BIOS-e820: 000000003feff000 - 000000003ff00000 (ACPI NVS)

……省略部分内容
```

显示开机信息

```
#pwd   //查看当前所在目录
/home/hnlinux/
# dmesg > boot.msg //将开机信息保存到 boot.msg文件中
#ls //显示当前目录文件
boot.msg
```

Linux depmod命令

Linux depmod命令用于分析可载入模块的相依性。

depmod(depend module)可检测模块的相依性，供modprobe在安装模块时使用。

语法

```
depmod [-adeisvV][-m <文件>][--help][模块名称]
```

参数说明：

- -a或--all 　分析所有可用的模块。
- -d或debug 　执行排错模式。
- -e 　输出无法参照的符号。
- -i 　不检查符号表的版本。
- -m<文件>或system-map<文件> 　使用指定的符号表文件。
- -s或--system-log 　在系统记录中记录错误。
- -v或--verbose 　执行时显示详细的信息。
- -V或--version 　显示版本信息。
- --help 　显示帮助。

实例

显示可用模块

```
# depmod -a //显示可用模块
```

Linux declare命令

Linux declare命令用于声明 shell 变量。

declare为shell指令，在第一种语法中可用来声明变量并设置变量的属性([rix]即为变量的属性），在第二种语法中可用来显示shell函数。若不加上任何参数，则会显示全部的shell变量与函数(与执行set指令的效果相同)。

语法

```
declare [+/-][rxi][变量名称＝设置值] 或 declare -f
```

参数说明：

- +/- "-"可用来指定变量的属性，"+"则是取消变量所设的属性。
- -f 仅显示函数。
- r 将变量设置为只读。
- x 指定的变量会成为环境变量，可供shell以外的程序来使用。
- i [设置值]可以是数值，字符串或运算式。

实例

声明整数型变量

```
# declare -i ab //声明整数型变量
# ab=56 //改变变量内容
# echo $ab //显示变量内容
56
```

改变变量属性

```
# declare -i ef //声明整数型变量
# ef=1  //变量赋值（整数值）
# echo $ef //显示变量内容
1
# ef="wer" //变量赋值（文本值）
# echo $ef
0
# declare +i ef //取消变量属性
# ef="wer"
# echo $ef
wer
```

设置变量只读

```
# declare -r ab //设置变量为只读
# ab=88 //改变变量内容
-bash: ab: 只读变量
# echo $ab //显示变量内容
56
```

声明数组变量

```
# declare -a cd='([0]="a" [1]="b" [2]="c")' //声明数组变量
# echo ${cd[1]}
b //显示变量内容

# echo ${cd[@]} //显示整个数组变量内容
a b c
```

显示函数

```
# declare -f
command_not_found_handle ()
{
  if [ -x /usr/lib/command-not-found ]; then
    /usr/bin/python /usr/lib/command-not-found -- $1;
    return $?;
  else
    if [ -x /usr/share/command-not-found ]; then
      /usr/bin/python /usr/share/command-not-found -- $1;
      return $?;
    else
      return 127;
    fi;
  fi
}
```

Linux crontab命令

Linux crontab是用来定期执行程序的命令。

当安装完成操作系统之后，默认便会启动此任务调度命令。

crond命令每分鍾会定期检查是否有要执行的工作，如果有要执行的工作便会自动执行该工作。

而linux任务调度的工作主要分为以下两类：

- 1、系统执行的工作：系统周期性所要执行的工作，如备份系统数据、清理缓存
- 2、个人执行的工作：某个用户定期要做的工作，例如每隔10分钟检查邮件服务器是否有新信，这些工作可由每个用户自行设置

语法

```
crontab [ -u user ] file
```

或

```
crontab [ -u user ] { -l | -r | -e }
```

说明：

crontab 是用来让使用者在固定时间或固定间隔执行程序之用，换句话说，也就是类似使用者的时程表。

-u user 是指设定指定 user 的时程表，这个前提是你必须要有其权限(比如说是 root)才能够指定他人的时程表。如果不使用 -u user 的话，就是表示设定自己的时程表。

参数说明：

- -e : 执行文字编辑器来设定时程表，内定的文字编辑器是 VI，如果你想用别的文字编辑器，则请先设定 VISUAL 环境变数来指定使用那个文字编辑器(比如说 setenv VISUAL joe)
- -r : 删除目前的时程表
- -l : 列出目前的时程表

时程表的格式如下：

```
f1 f2 f3 f4 f5 program
```

- 其中 f1 是表示分钟，f2 表示小时，f3 表示一个月份中的第几日，f4 表示月份，f5 表示一个星期中的第几天。program 表示要执行的程序。
- 当 f1 为 时表示每分钟都要执行 *program，f2 为* 时表示每小时都要执行程序，其馀类推
- 当 f1 为 a-b 时表示从第 a 分钟到第 b 分钟这段时间内要执行，f2 为 a-b 时表示从第 a 到第 b 小时都要执行，其馀类推
- 当 f1 为 */n* 时表示每 *n* 分钟个时间间隔执行一次，*f2* 为 /n 表示每 n 小时个时间间隔执行一次，其馀类推
- 当 f1 为 a, b, c,... 时表示第 a, b, c,... 分钟要执行，f2 为 a, b, c,... 时表示第 a, b, c...个小时要执行，其馀类推

使用者也可以将所有的设定先存放在文件中，用 crontab file 的方式来设定时程表。

实例

每月每天每小时的第 0 分钟执行一次 /bin/ls

```
0 7 * * * /bin/ls
```

在 12 月内, 每天的早上 6 点到 12 点中，每隔 20 分钟执行一次 /usr/bin/backup

```
0 6-12/3 * 12 * /usr/bin/backup
```

周一到周五每天下午 5:00 寄一封信给 alex@domain.name

```
0 17 * * 1-5 mail -s "hi" alex@domain.name < /tmp/maildata
```

每月每天的午夜 0 点 20 分, 2 点 20 分, 4 点 20 分....执行 echo "haha"

```
20 0-23/2 * * * echo "haha"
```

下面再看看几个具体的例子：

```
0 */2 * * * /sbin/service httpd restart  意思是每两个小时重启一次apache

50 7 * * * /sbin/service sshd start  意思是每天7：50开启ssh服务

50 22 * * * /sbin/service sshd stop  意思是每天22：50关闭ssh服务

0 0 1,15 * * fsck /home  每月1号和15号检查/home 磁盘

1 * * * * /home/bruce/backup  每小时的第一分执行 /home/bruce/backup这个文件

00 03 * * 1-5 find /home "*.xxx" -mtime +4 -exec rm {} \;  每周一至周五3点钟，在目录/home中，

30 6 */10 * * ls  意思是每月的1、11、21、31日是的6：30执行一次ls命令
```

注意：当程序在你所指定的时间执行后，系统会寄一封信给你，显示该程序执行的内容，若是你不希望收到这样的信，请在每一行空一格之后加上 > /dev/null 2>&1 即可

Linux clock命令

Linux clock命令用于调整 RTC 时间。

RTC 是电脑内建的硬件时间，执行这项指令可以显示现在时刻，调整硬件时钟的时间，将系统时间设成与硬件时钟之时间一致，或是把系统时间回存到硬件时钟。

语法

```
clock [--adjust][--debug][--directisa][--getepoch][--hctosys][--set --date="<日期时间>"][-
```

参数说明：

- --adjust　第一次使用"--set"或"--systohc"参数设置硬件时钟，会在/etc目录下产生一个名称为adjtime的文件。当再次使用这两个参数调整硬件时钟，此文件便会记录两次调整间之差异，日后执行clock指令加上"--adjust"参数时，程序会自动根 据记录文件的数值差异，计算出平均值，自动调整硬件时钟的时间。
- --debug　详细显示指令执行过程，便于排错或了解程序执行的情形。
- --directisa　告诉clock指令不要通过/dev/rtc设备文件，直接对硬件时钟进行存取。这个参数适用于仅有ISA总线结构的老式电脑。
- --getepoch　把系统核心内的硬件时钟新时代数值，呈现到标准输出设备。
- --hctosys　Hardware Clock to System Time，把系统时间设成和硬件时钟一致。由于这个动作将会造成系统全面更新文件的存取时间，所以最好在系统启动时就执行它。
- --set--date　设置硬件时钟的日期和时间。
- --setepoch--epoch=<年份>　设置系统核心之硬件时钟的新时代数值，年份以四位树字表示。
- --show　读取硬件时钟的时间，并将其呈现至标准输出设备。
- --systohc　System Time to Hardware Clock，将系统时间存回硬件时钟内。
- --test　仅作测试，并不真的将时间写入硬件时钟或系统时间。
- --utc　把硬件时钟上的时间时为CUT，有时也称为UTC或UCT。
- --version　显示版本信息。

实例

获取当前的时间

```
# clock //获取当前的时间
```

显示UTC时间

```
# clock -utc //显示UTC时间
```

Linux chroot命令

Linux chroot命令用于改变根目录。

chroot(change root)命令把根目录换成指定的目的目录。

、

语法

```
chroot [--help][--version][目的目录][执行指令...]
```

参数说明：

- --help　在线帮助。
- --version　显示版本信息。

实例

改变根目录

```
# chroot /mnt/ls //改变根目录
```

Linux insmod命令

Linux insmod(install module)命令用于载入模块。

Linux有许多功能是通过模块的方式，在需要时才载入kernel。如此可使kernel较为精简，进而提高效率，以及保有较大的弹性。这类可载入的模块，通常是设备驱动程序。

语法

```
insmod [-fkmpsvxX][-o <模块名称>][模块文件][符号名称 = 符号值]
```

参数说明：

- -f 　不检查目前kernel版本与模块编译时的kernel版本是否一致，强制将模块载入。
- -k 　将模块设置为自动卸除。
- -m 　输出模块的载入信息。
- -o<模块名称> 　指定模块的名称，可使用模块文件的文件名。
- -p 　测试模块是否能正确地载入kernel。
- -s 　将所有信息记录在系统记录文件中。
- -v 　执行时显示详细的信息。
- -x 　不要汇出模块的外部符号。
- -X 　汇出模块所有的外部符号，此为预设置。

实例

加载模块

```
# insmod led.o
//向内核加载模块
```

Linux rpm命令

Linux rpm命令用于管理套件。

rmp(redhat package manager)原本是Red Hat Linux发行版专门用来管理Linux各项套件的程序，由于它遵循GPL规则且功能强大方便，因而广受欢迎。逐渐受到其他发行版的采用。RPM套件管理方式的出现，让Linux易于安装，升级，间接提升了Linux的适用度。

语法

```
rpm [-acdhilqRsv][-b<完成阶段><套间档>+][-e<套件挡>][-f<文件>+][-i<套件档>][-p<套件档>＋][-U<套件
```

参数说明：

- -a 　查询所有套件。
- -b<完成阶段><套件档>+或-t <完成阶段><套件档>+ 　设置包装套件的完成阶段，并指定套件档的文件名称。
- -c 　只列出组态配置文件，本参数需配合"-l"参数使用。
- -d 　只列出文本文件，本参数需配合"-l"参数使用。
- -e<套件档>或--erase<套件档> 　删除指定的套件。
- -f<文件>+ 　查询拥有指定文件的套件。
- -h或--hash 　套件安装时列出标记。
- -i 　显示套件的相关信息。
- -i<套件档>或--install<套件档> 　安装指定的套件档。
- -l 　显示套件的文件列表。
- -p<套件档>+ 　查询指定的RPM套件档。
- -q 　使用询问模式，当遇到任何问题时，rpm指令会先询问用户。
- -R 　显示套件的关联性信息。
- -s 　显示文件状态，本参数需配合"-l"参数使用。
- -U<套件档>或--upgrade<套件档> 升级指定的套件档。
- -v 　显示指令执行过程。
- -vv 　详细显示指令执行过程，便于排错。
- -addsign<套件档>+ 　在指定的套件里加上新的签名认证。
- --allfiles 　安装所有文件。
- --allmatches 　删除符合指定的套件所包含的文件。
- --badreloc 　发生错误时，重新配置文件。
- --buildroot<根目录> 　设置产生套件时，欲当作根目录的目录。
- --changelog 　显示套件的更改记录。

- --checksig<套件档>+　检验该套件的签名认证。
- --clean　完成套件的包装后，删除包装过程中所建立的目录。
- --dbpath<数据库目录>　设置欲存放RPM数据库的目录。
- --dump　显示每个文件的验证信息。本参数需配合"-l"参数使用。
- --excludedocs　安装套件时，不要安装文件。
- --excludepath<排除目录>　忽略在指定目录里的所有文件。
- --force　强行置换套件或文件。
- --ftpproxy<主机名称或IP地址>　指定FTP代理服务器。
- --ftpport<通信端口>　设置FTP服务器或代理服务器使用的通信端口。
- --help　在线帮助。
- --httpproxy<主机名称或IP地址>　指定HTTP代理服务器。
- --httpport<通信端口>　设置HTTP服务器或代理服务器使用的通信端口。
- --ignorearch　不验证套件档的结构正确性。
- --ignoreos　不验证套件档的结构正确性。
- --ignoresize　安装前不检查磁盘空间是否足够。
- --includedocs　安装套件时，一并安装文件。
- --initdb　确认有正确的数据库可以使用。
- --justdb　更新数据库，当不变动任何文件。
- --nobulid　不执行任何完成阶段。
- --nodeps　不验证套件档的相互关联性。
- --nofiles　不验证文件的属性。
- --nogpg　略过所有GPG的签名认证。
- --nomd5　不使用MD5编码演算确认文件的大小与正确性。
- --nopgp　略过所有PGP的签名认证。
- --noorder　不重新编排套件的安装顺序，以便满足其彼此间的关联性。
- --noscripts　不执行任何安装Script文件。
- --notriggers　不执行该套件包装内的任何Script文件。
- --oldpackage　升级成旧版本的套件。
- --percent　安装套件时显示完成度百分比。
- --pipe<执行指令>　建立管道，把输出结果转为该执行指令的输入数据。
- --prefix<目的目录>　若重新配置文件，就把文件放到指定的目录下。
- --provides　查询该套件所提供的兼容度。
- --queryformat<档头格式>　设置档头的表示方式。
- --querytags　列出可用于档头格式的标签。
- --rcfile<配置文件>　使用指定的配置文件。
- --rebulid<套件档>　安装原始代码套件，重新产生二进制文件的套件。
- --rebuliddb　以现有的数据库为主，重建一份数据库。
- --recompile<套件档>　此参数的效果和指定"--rebulid"参数类似，当不产生套件档。
- --relocate<原目录>=<新目录>　把本来会放到原目录下的文件改放到新目录。
- --replacefiles　强行置换文件。

- --replacepkgs　强行置换套件。
- --requires　查询该套件所需要的兼容度。
- --resing<套件档>+　删除现有认证，重新产生签名认证。
- --rmsource　完成套件的包装后，删除原始代码。
- --rmsource<文件>　删除原始代码和指定的文件。
- --root<根目录>　设置欲当作根目录的目录。
- --scripts　列出安装套件的Script的变量。
- --setperms　设置文件的权限。
- --setugids　设置文件的拥有者和所属群组。
- --short-circuit　直接略过指定完成阶段的步骤。
- --sign　产生PGP或GPG的签名认证。
- --target=<安装平台>+　设置产生的套件的安装平台。
- --test　仅作测试，并不真的安装套件。
- --timecheck<检查秒数>　设置检查时间的计时秒数。
- --triggeredby<套件档>　查询该套件的包装者。
- --triggers　展示套件档内的包装Script。
- --verify　此参数的效果和指定"-q"参数相同。
- --version　显示版本信息。
- --whatprovides<功能特性>　查询该套件对指定的功能特性所提供的兼容度。
- --whatrequires<功能特性>　查询该套件对指定的功能特性所需要的兼容度。

实例

安装软件

```
# rpm -hvi dejagnu-1.4.2-10.noarch.rpm
警告：dejagnu-1.4.2-10.noarch.rpm: V3 DSA 签名：NOKEY, key ID db42a60e
准备...
########################################### [100%]
```

显示软件安装信息

```
# rpm -qi dejagnu-1.4.2-10.noarch.rpm

【第1次更新 教程、类似命令关联】
```

Linux grpconv命令

Linux grpconv(group convert to shadow password)命令用于开启群组的投影密码。

Linux系统里的用户和群组密码，分别存放在/etc目录下的passwd和group文件中。因系统运作所需，任何人都得以读取它们，造成安全上的破绽。投影密码将文件内的密码改存在/etc目录下的shadow和gshadow文件内，只允许系统管理者读取，同时把原密码置换为"x"字符。投影密码的功能可随时开启或关闭，您只需执行grpconv指令就能开启群组投影密码。

语法

```
grpconv
```

Linux pwunconv命令

Linux pwunconv命令用于关闭用户的投影密码。

执行pwunconv指令可以关闭用户投影密码，它会把密码从shadow文件内，重回存到passwd文件里。

语法

```
pwunconv
```

实例

关闭用户的投影密码

```
# pwunconv
```

Linux export命令

Linux export命令用于设置或显示环境变量。

在shell中执行程序时，shell会提供一组环境变量。export可新增，修改或删除环境变量，供后续执行的程序使用。export的效力仅及于该次登陆操作。

语法

```
export [-fnp][变量名称]=[变量设置值]
```

参数说明：

- -f　代表[变量名称]中为函数名称。
- -n　删除指定的变量。变量实际上并未删除，只是不会输出到后续指令的执行环境中。
- -p　列出所有的shell赋予程序的环境变量。

实例

列出当前所有的环境变量

```
# export -p //列出当前的环境变量值
declare -x HOME="/root"
declare -x LANG="zh_CN.UTF-8"
declare -x LANGUAGE="zh_CN:zh"
declare -x LESSCLOSE="/usr/bin/lesspipe %s %s"
declare -x LESSOPEN="| /usr/bin/lesspipe %s"
declare -x LOGNAME="root"
declare -x LS_COLORS=""
declare -x MAIL="/var/mail/root"
declare -x OLDPWD
declare -x PATH="/opt/toolchains/arm920t-eabi/bin:/opt/toolchains/arm920t-eabi/bin:/usr/l
declare -x PWD="/root"
declare -x SHELL="/bin/bash"
declare -x SHLVL="1"
declare -x SPEECHD_PORT="6560"
declare -x SSH_CLIENT="192.168.1.65 1674 22"
declare -x SSH_CONNECTION="192.168.1.65 1674 192.168.1.3 22"
declare -x SSH_TTY="/dev/pts/2"
declare -x TERM="XTERM"
declare -x USER="root"
declare -x XDG_SESSION_COOKIE="93b5d3d03e032c0cf892a4474bebda9f-1273864738.954257-3402064
```

定义环境变量

```
# export MYENV //定义环境变量
# export -p //列出当前的环境变量
declare -x HOME="/root"
declare -x LANG="zh_CN.UTF-8"
declare -x LANGUAGE="zh_CN:zh"
declare -x LESSCLOSE="/usr/bin/lesspipe %s %s"
declare -x LESSOPEN="| /usr/bin/lesspipe %s"
declare -x LOGNAME="root"
declare -x LS_COLORS=""
declare -x MAIL="/var/mail/root"
declare -x MYENV
declare -x OLDPWD
declare -x PATH="/opt/toolchains/arm920t-eabi/bin:/opt/toolchains/arm920t-eabi/bin:/usr/l
declare -x PWD="/root"
declare -x SHELL="/bin/bash"
declare -x SHLVL="1"
declare -x SPEECHD_PORT="6560"
declare -x SSH_CLIENT="192.168.1.65 1674 22"
declare -x SSH_CONNECTION="192.168.1.65 1674 192.168.1.3 22"
declare -x SSH_TTY="/dev/pts/2"
declare -x TERM="XTERM"
declare -x USER="root"
declare -x XDG_SESSION_COOKIE="93b5d3d03e032c0cf892a4474bebda9f-1273864738.954257-3402064
```

定义环境变量赋值

```
# export MYENV=7 //定义环境变量并赋值
# export -p
declare -x HOME="/root"
declare -x LANG="zh_CN.UTF-8"
declare -x LANGUAGE="zh_CN:zh"
declare -x LESSCLOSE="/usr/bin/lesspipe %s %s"
declare -x LESSOPEN="| /usr/bin/lesspipe %s"
declare -x LOGNAME="root"
declare -x LS_COLORS=""
declare -x MAIL="/var/mail/root"
declare -x MYENV="7"
declare -x OLDPWD
declare -x PATH="/opt/toolchains/arm920t-eabi/bin:/opt/toolchains/arm920t-eabi/bin:/usr/l
declare -x PWD="/root"
declare -x SHELL="/bin/bash"
declare -x SHLVL="1"
declare -x SPEECHD_PORT="6560"
declare -x SSH_CLIENT="192.168.1.65 1674 22"
declare -x SSH_CONNECTION="192.168.1.65 1674 192.168.1.3 22"
declare -x SSH_TTY="/dev/pts/2"
declare -x TERM="XTERM"
declare -x USER="root"
declare -x XDG_SESSION_COOKIE="93b5d3d03e032c0cf892a4474bebda9f-1273864738.954257-3402064
```

Linux eval命令

Linux eval命令用于重新运算求出参数的内容。

eval可读取一连串的参数，然后再依参数本身的特性来执行。

语法

```
eval [参数]
```

参数说明：参数不限数目，彼此之间用分号分开。

实例

连接多个命令

```
# eval enable;ls //连接多个命令
enable .
enable :
enable [
enable alias
enable bg
enable bind
enable break
enable builtin
enable caller
enable cd
enable command
enable compgen
enable complete
enable compopt
enable continue
enable declare
enable dirs
enable disown
enable echo
enable enable
enable eval
enable exec
enable exit
enable export
enable false
enable fc
enable fg
enable getopts
enable hash
enable help
enable history
enable jobs
enable kill
enable let
enable local
enable logout
enable mapfile
enable popd
enable printf
enable pushd
enable pwd
enable read
enable readarray
enable readonly
enable return
enable set
enable shift
enable shopt
enable source
enable suspend
enable test
enable times
enable trap
enable true
enable type
enable typeset
enable ulimit
enable umask
enable unalias
enable unset
enable wait
```

Linux set命令

Linux set命令用于设置shell。

set指令能设置所使用shell的执行方式，可依照不同的需求来做设置。

语法

```
set [+-abCdefhHklmnpPtuvx]
```

参数说明：

- -a　标示已修改的变量，以供输出至环境变量。
- -b　使被中止的后台程序立刻回报执行状态。
- -C　转向所产生的文件无法覆盖已存在的文件。
- -d　Shell预设会用杂凑表记忆使用过的指令，以加速指令的执行。使用-d参数可取消。
- -e　若指令传回值不等于0，则立即退出shell。
- -f　取消使用通配符。
- -h　自动记录函数的所在位置。
- -H Shell　可利用"!"加<指令编号>的方式来执行history中记录的指令。
- -k　指令所给的参数都会被视为此指令的环境变量。
- -l　记录for循环的变量名称。
- -m　使用监视模式。
- -n　只读取指令，而不实际执行。
- -p　启动优先顺序模式。
- -P　启动-P参数后，执行指令时，会以实际的文件或目录来取代符号连接。
- -t　执行完随后的指令，即退出shell。
- -u　当执行时使用到未定义过的变量，则显示错误信息。
- -v　显示shell所读取的输入值。
- -x　执行指令后，会先显示该指令及所下的参数。
- +<参数>　取消某个set曾启动的参数。

实例

显示环境变量

```
# set
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="3" [1]="00" [2]="15" [3]="1" [4]="release" [5]="i386-redhat-linux-gnu
BASH_VERSION='3.00.15(1)-release'
COLORS=/etc/DIR_COLORS.xterm
COLUMNS=99
DIRSTACK=()
EUID=0
GROUPS=()
G_BROKEN_FILENAMES=1
HISTFILE=/root/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/root
HOSTNAME=hnlinux
HOSTTYPE=i386
IFS=/plinux> '
INPUTRC=/etc/inputrc
KDEDIR=/usr
LANG=zh_CN.GB2312
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=34
L
MAIL=/var/spool/mail/root
MAILCHECK=60
OLDPWD=/home/uptech
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PATH=/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/
PIPESTATUS=([0]="2")
PPID=26005
PROMPT_COMMAND='echo -ne "
```

Linux minfo命令

Linux minfo命令用于显示MS-DOS文件系统的各项参数。

minfo为mtools工具指令，可显示MS-DOS系统磁盘的各项参数，包括磁区数，磁头数...等。

语法

```
</>minfo [-v][驱动器代号]
```

参数说明：

- -v　除了一般信息外，并显示可开机磁区的内容。

实例

显示DOS系统参数

```
# minfo -v C: //显示系统参数
```

Linux lsmod命令

Linux lsmod命令用于显示已载入系统的模块。

执行lsmod(list modules)指令，会列出所有已载入系统的模块。Linux操作系统的核心具有模块化的特性，应此在编译核心时，务须把全部的功能都放入核心。您可以将这些功能编译成一个个单独的模块，待需要时再分别载入。

语法

```
lsmod
```

实例

显示模块信息

```
# lsmod
Module         Size Used by
nfsd          238935 11
lockd          64849 1 nfsd
nfs_acl          2245 1 nfsd
auth_rpcgss       33735 1 nfsd
sunrpc         193181 10 nfsd,lockd,nfs_acl,auth_rpcgss
exportfs         3437 1 nfsd
xt_TCPMSS         2931 1
xt_tcpmss         1197 1
xt_tcpudp         2011 1
iptable_mangle      2771 1
ip_tables         9991 1 iptable_mangle
x_tables         14299 4 xt_TCPMSS,xt_tcpmss,xt_tcpudp,ip_tables
pppoe           8943 2
pppox           2074 1 pppoe
binfmt_misc        6587 1
snd_ens1371       18814 0
gameport         9089 1 snd_ens1371
snd_ac97_codec     100646 1 snd_ens1371
ac97_bus         1002 1 snd_ac97_codec
snd_pcm_oss       35308 0
snd_mixer_oss      13746 1 snd_pcm_oss
snd_pcm         70662 3 snd_ens1371,snd_ac97_codec,snd_pcm_oss
snd_seq_dummy       1338 0
snd_seq_oss       26726 0
snd_seq_midi       4557 0
snd_rawmidi       19056 2 snd_ens1371,snd_seq_midi
snd_seq_midi_event   6003 2 snd_seq_oss,snd_seq_midi
snd_seq         47263 6 snd_seq_dummy,snd_seq_oss,snd_seq_midi,snd_seq_midi_event
snd_timer        19098 2 snd_pcm,snd_seq
snd_seq_device      5700 5 snd_seq_dummy,snd_seq_oss,snd_seq_midi,snd_rawmidi,snd_seq
fbcon          35102 71
tileblit         2031 1 fbcon
font           7557 1 fbcon
bitblit          4707 1 fbcon
ppdev           5259 0
softcursor        1189 1 bitblit
snd           54148 10 snd_ens1371,snd_ac97_codec,snd_pcm_oss,snd_mixer_oss,snd_pcm,snd_se
psmouse         63245 0
serio_raw         3978 0
soundcore         6620 1 snd
parport_pc        25962 1
snd_page_alloc      7076 1 snd_pcm
vga16fb         11385 1
intel_agp        24177 1
vgastate         8961 1 vga16fb
i2c_piix4         8335 0
shpchp          28820 0
agpgart         31724 1 intel_agp
lp           7028 0
parport          32635 3 ppdev,parport_pc,lp
mptspi          14652 2
mptscsih         31325 1 mptspi
pcnet32         28890 0
floppy          53016 0
mii           4381 1 pcnet32
mptbase         83022 2 mptspi,mptscsih
scsi_transport_spi   21096 1 mptspi
```

Linux liloconfig命令

Linux liloconfig命令用于设置核心载入，开机管理程序。

liloconfig是Slackware发行版专门用来调整lilo设置的程序。它通过互动式操作界面，让用户能够利用键盘上的方向键等，轻易地操控lilo的安装，设置作业，而无须下达各种参数或撰写配置文件。

语法

```
liloconfig
```

实例

执行liloconfig命令

```
# liloconfig
```

Linux lilo命令

Linux lilo命令用于安装核心载入，开机管理程序。

lilo(linux loader)是个Linux系统核心载入程序，同时具备管理开机的功能。单独执行lilo指令，它会读取/etc/目录下的lilo.conf配置文件，然后根据其内容安装lilo。

语法

```
lilo [-clqtV][-b<外围设备代号>][-C<配置文件>][-d<延迟时间>][－D<识别标签>][-f<几何参数文件>][-i<开
```

参数说明：

- -b<外围设备代号>　指定安装lilo之处的外围设备代号。
- -c　使用紧致映射模式。
- -C<配置文件>　指定lilo的配置文件。
- -d<延迟时间>　设置开机延迟时间。
- -D<识别标签>　指定开机后预设启动的操作系统，或系统核心识别标签。
- -f<几何参数文件>　指定磁盘的几何参数配置文件。
- -i<开机磁区文件>　指定欲使用的开机磁区文件，预设是/boot目录里的boot.b文件。
- -I<识别标签>　显示系统核心存放之处。
- -l　产生线形磁区地址。
- -m<映射文件>　指定映射文件。
- -P<fix/ignore>　决定要修复或忽略分区表的错误。
- -q　列出映射的系统核心文件。
- -r<根目录>　设置系统启动时欲挂入成为根目录的目录。
- -R<执行指令>　设置下次启动系统时，首先执行的指令。
- -s<备份文件>　指定备份文件。
- -S<备份文件>　强制指定备份文件。
- -t　不执行指令，仅列出实际执行会进行的动作。
- -u<外围色设备代号>　删除lilo。
- -U<外围设备代号>　此参数的效果和指定"-u"参数类似，当不检查时间戳记。
- -v　显示指令执行过程。
- -V　显示版本信息。

实例

安装lilo到第一台SCSI硬盘的第三个主要分区，采用3级模式。

```
# lilo -b /dev/sda3 -v -v -v
```

指定安装lilo的配置文件和备份文件。

```
# lilo -C /etc/lilo.conf2 -s /boot/boot. Backup
```

Linux kbdconfig命令

Linux kbdconfig命令用于设置键盘类型。

kbdconfig(Red Hat Linux才有的指令)是一个用来设置键盘的程序，提供图形化的操作界面。kbdconfig实际上是修改/etc/sysconfig/keyboard的键盘配置文件。

语法

```
kbdconfig [--back][--test]
```

参数：

- --back 　执行时将预设的Cancel按钮更改为Back按钮。
- --test 　仅作测试，不会实际更改设置。

实例

键盘设置：

```
# kdbconfig //设置键盘
```

Linux modprobe命令

Linux modprobe命令用于自动处理可载入模块。

modprobe可载入指定的个别模块，或是载入一组相依的模块。modprobe会根据depmod所产生的相依关系，决定要载入哪些模块。若在载入过程中发生错误，在modprobe会卸载整组的模块。

语法

```
modprobe [-acdlrtvV][--help][模块文件][符号名称 = 符号值]
```

参数：

- -a或--all 载入全部的模块。
- -c或--show-conf 显示所有模块的设置信息。
- -d或--debug 使用排错模式。
- -l或--list 显示可用的模块。
- -r或--remove 模块闲置不用时，即自动卸载模块。
- -t或--type 指定模块类型。
- -v或--verbose 执行时显示详细的信息。
- -V或--version 显示版本信息。
- -help 显示帮助。

实例

安装软驱模块：

```
[root@w3cschool.cc ~]# modprobe -v floppy
```

卸载软驱模块：

```
[root@w3cschool.cc ~]# modprobe -v -r floppy
```

Linux ntsysv命令

Linux ntsysv命令用于设置系统的各种服务。

这是Red Hat公司遵循GPL规则所开发的程序，它具有互动式操作界面，您可以轻易地利用方向键和空格键等，开启，关闭操作系统在每个执行等级中，所要执行的系统服务。

语法

```
ntsysv [--back][--level <等级代号>]
```

参数：

- --back 　在互动式界面里，显示Back钮，而非Cancel钮。
- --level <等级代号> 　在指定的执行等级中，决定要开启或关闭哪些系统服务。

Linux mouseconfig命令

Linux mouseconfig命令用于设置鼠标相关参数。

mouseconfig为鼠标设置程序，可自动设置相关参数，或者用户也可以利用所提供互动模式自行设置鼠标。mouseconfig是Red Hat Linux才有的命令。

语法

```
mouseconfig [--back][--emulthree][--help][--expert][--kickstart][--noprobe][--test][--dev
```

参数：

- --back　在设置画面上显示Back按钮，而取代预设的Cancel按钮。
- --device<连接端口>　指定硬件连接端口。可用的选项有ttyS0，ttyS1，ttyS2，ttyS3与orpsaux。
- --emulthree　将二钮鼠标模拟成三钮鼠标。
- --help　显示帮助以及所有支持的鼠标类型。
- --expert　程序预设可自动判断部分设置值。若要自行设置，请使用--expert参数。
- --kickstart　让程序自动检测并保存所有的鼠标设置。
- --noprobe　不要检测鼠标设备。
- --test　测试模式，不会改变任何设置。

实例

以交互模式配置鼠标：

```
# mouseconfig -text
```

Linux passwd命令

Linux passwd命令用来更改使用者的密码

语法

```
passwd [-k] [-l] [-u [-f]] [-d] [-S] [username]
```

必要参数：

- -d 删除密码
- -f 强制执行
- -k 更新只能发送在过期之后
- -l 停止账号使用
- -S 显示密码信息
- -u 启用已被停止的账户
- -x 设置密码的有效期
- -g 修改群组密码
- -i 过期后停止用户账号

选择参数：

- --help 显示帮助信息
- --version 显示版本信息

实例

修改用户密码

```
# passwd w3cschool  //设置w3cschool用户的密码
Enter new UNIX password:  //输入新密码，输入的密码无回显
Retype new UNIX password:  //确认密码
passwd: password updated successfully
#
```

显示账号密码信息

```
# passwd -S w3cschool
w3cschool P 05/13/2010 0 99999 7 -1
```

删除用户密码

```
# passwd -d lx138
passwd: password expiry information changed.
```

Linux pwconv命令

Linux pwconv命令用于开启用户的投影密码。

Linux系统里的用户和群组密码，分别存放在名称为passwd和group的文件中， 这两个文件位于/etc目录下。因系统运作所需，任何人都得以读取它们，造成安全上的破绽。投影密码将文件内的密码改存在/etc目录下的shadow和gshadow文件内，只允许系统管理者读取，同时把原密码置换为"x"字符，有效的强化了系统的安全性。

语法

```
pwconv
```

实例

开启用户的投影密码

```
# pwconv
```

Linux rdate命令

Linux rdate命令用于显示其他主机的日期与时间。

执行rdate指令，向其他主机询问系统时间并显示出来。

语法

```
rdate [-ps][主机名称或IP地址...]
```

参数：

- -p 　显示远端主机的日期与时间。
- -s 　把从远端主机收到的日期和时间，回存到本地主机的系统时间。
- -u 传输协议使用UDP协议
- -l 使用syslog显示错误信息
- -t<时间> 设置超时时间

Linux resize命令

Linux resize命令设置终端机视窗的大小。

执行resize指令可设置虚拟终端机的视窗大小。

语法

```
resize [-cu][-s <列数> <行数>]
```

参数：

- -c 　就算用户环境并非C Shell，也用C Shell指令改变视窗大小。
- -s <列数> <行数> 　设置终端机视窗的垂直高度和水平宽度。
- -u 　就算用户环境并非Bourne Shell，也用Bourne Shell指令改变视窗大小。

实例

使用 C shell

```
[root@linux w3cschool.cc]# resize -c
set noglob;
setenv COLUMNS '99';
setenv LINES '34';
unset noglob;
```

使用 Bourne shell

```
[root@hnlinux w3cschool.cc]# resize -u
COLUMNS=99;
LINES=34;
export COLUMNS LINES;
```

设置指定大小

```
[root@hnlinux w3cschool.cc]# resize -s 80 160
```

Linux rmmod命令

Linux rmmod命令用于删除模块。

执行rmmod指令，可删除不需要的模块。Linux操作系统的核心具有模块化的特性，应此在编译核心时，务须把全部的功能都放如核心。你可以将这些功能编译成一个个单独的模块，待有需要时再分别载入它们。

语法

```
rmmod [-as][模块名称...]
```

参数：

- -a 　删除所有目前不需要的模块。
- -s 　把信息输出至syslog常驻服务，而非终端机界面。

实例

显示已安装的模块

```
# lsmod
Module         Size Used by
cramfs         39042 1
nfsd          238935 11
lockd          64849 1 nfsd
nfs_acl         2245 1 nfsd
auth_rpcgss      33735 1 nfsd
sunrpc         193181 10 nfsd,lockd,nfs_acl,auth_rpcgss
exportfs        3437 1 nfsd
xt_TCPMSS        2931 0
xt_tcpmss        1197 0
xt_tcpudp        2011 0
iptable_mangle     2771 0
ip_tables        9991 1 iptable_mangle
x_tables        14299 4

……省略部分结果
pppoe          8943 0
pppox          2074 1 pppoe
binfmt_misc       6587 1
snd_ens1371      18814 0
gameport        9089 1 snd_ens1371
snd_ac97_codec    100646 1 snd_ens1371
ac97_bus        1002 1 snd_ac97_codec
snd_pcm_oss      35308 0
```

卸载模块

```
# rmmod -v pppoe //卸载模块pppoe
Checking ppoe for persistent data
```

安装模块

```
# insmod -v pppoe >1.log //安装模块

~# tail -b 30 1.log //显示文件信息
```

Linux grpunconv命令

Linux grpunconv命令用于关闭群组的投影密码。

执行grpunconv指令可关闭群组投影密码，它会把密码从gshadow文件内，回存到group文件里。

语法

```
grpunconv
```

实例

未关闭的情况

```
cat /etc/gshadow | grep cdy
cdy:123456::
```

关闭影子密码

```
cat /etc/gshadow
cat: /etc/gshadow: 没有那个文件或目录
```

查看密码已经复制到 /etc/group 中了。

```
cat /etc/group | grep cdy
cdy:123456:1000:
```

Linux modinfo命令

Linux modinfo命令用于显示kernel模块的信息。

modinfo会显示kernel模块的对象文件，以显示该模块的相关信息。

语法

```
modinfo [-adhpV][模块文件]
```

参数：

- -a或--author 　显示模块开发人员。
- -d或--description 　显示模块的说明。
- -h或--help 　显示modinfo的参数使用方法。
- -p或--parameters 　显示模块所支持的参数。
- -V或--version 　显示版本信息。

实例

显示sg模块的信息。

```
# modinfo sg
filename:    /lib/modules/2.6.9-42.ELsmp/kernel/drivers/scsi/sg.ko
author:     Douglas Gilbert
description:  SCSI generic (sg) driver
license:    GPL
version:    3.5.31 B0B0CB1BB59F0669A1F0D6B
parm:      def_reserved_size:size of buffer reserved for each fd
parm:      allow_dio:allow direct I/O (default: 0 (disallow))
alias:     char-major-21-*
vermagic:    2.6.9-42.ELsmp SMP 686 REGPARM 4KSTACKS gcc-3.4
depends:    scsi_mod
```

Linux time命令

Linux time命令的用途，在于量测特定指令执行时所需消耗的时间及系统资源等资讯。

例如 CPU 时间、记忆体、输入输出等等。需要特别注意的是，部分资讯在 Linux 上显示不出来。这是因为在 Linux 上部分资源的分配函式与 time 指令所预设的方式并不相同，以致于 time 指令无法取得这些资料。

语法

```
time [options] COMMAND [arguments]
```

参数：

- -o 或 --output=FILE：设定结果输出档。这个选项会将 time 的输出写入 所指定的档案中。如果档案已经存在，系统将覆写其内容。
- -a 或 --append：配合 -o 使用，会将结果写到档案的末端，而不会覆盖掉原来的内容。
- -f FORMAT 或 --format=FORMAT：以 FORMAT 字串设定显示方式。当这个选项没有被设定的时候，会用系统预设的格式。不过你可以用环境变数 time 来设定这个格式，如此一来就不必每次登入系统都要设定一次。

time 指令可以显示的资源有四大项，分别是：

- Time resources
- Memory resources
- IO resources
- Command info

详细的内容如下：

1、Time Resources

E 执行指令所花费的时间，格式是：[hour]:minute:second。请注意这个数字并不代表实际的 CPU 时间。

e 执行指令所花费的时间，单位是秒。请注意这个数字并不代表实际的 CPU 时间。

S 指令执行时在核心模式（kernel mode）所花费的时间，单位是秒。

U 指令执行时在使用者模式（user mode）所花费的时间，单位是秒。

P 执行指令时 CPU 的占用比例。其实这个数字就是核心模式加上使用者模式的 CPU 时间除以总时间。

2、Memory Resources

M 执行时所占用的实体记忆体的最大值。单位是 KB

t 执行时所占用的实体记忆体的平均值，单位是 KB

K 执行程序所占用的记忆体总量（stack+data+text）的平均大小，单位是 KB

D 执行程序的自有资料区（unshared data area）的平均大小，单位是 KB

p 执行程序的自有堆叠（unshared stack）的平均大小，单位是 KB

X 执行程序间共享内容（shared text）的平均值，单位是 KB

Z 系统记忆体页的大小，单位是 byte。对同一个系统来说这是个常数

3、IO Resources

F 此程序的主要记忆体页错误发生次数。所谓的主要记忆体页错误是指某一记忆体页已经置换到置换档（swap file)中，而且已经分配给其他程序。此时该页的内容必须从置换档里再读出来。

R 此程序的次要记忆体页错误发生次数。所谓的次要记忆体页错误是指某一记忆体页虽然已经置换到置换档中，但尚未分配给其他程序。此时该页的内容并未被破坏，不必从置换档里读出来

W 此程序被交换到置换档的次数

c 此程序被强迫中断（像是分配到的 CPU 时间耗尽）的次数

w 此程序自愿中断（像是在等待某一个 I/O 执行完毕，像是磁碟读取等等）的次数

I 此程序所输入的档案数

O 此程序所输出的档案数

r 此程序所收到的 Socket Message

s 此程序所送出的 Socket Message

k 此程序所收到的信号 (Signal)数量

4、Command Info

C 执行时的参数以及指令名称

x 指令的结束代码 (Exit Status)

-p or --portability：这个选项会自动把显示格式设定成为：

real %e user %Usys %S：这么做的目的是为了与 POSIX 规格相容。

-v or --verbose：这个选项会把所有程序中用到的资源通通列出来，不但如一般英文语句，还有说明。对不想花时间去熟习格式设定或是刚刚开始接触这个指令的人相当有用。

实例

```
1\. # time date
2\. Sun Mar 26 22:45:34 GMT-8 2006
3\.
4\. real    0m0.136s
5\. user    0m0.010s
6\. sys     0m0.070s
7\. #
```

在以上实例中，执行命令"time date"(见第1行)。

系统先执行命令"date"，第2行为命令"date"的执行结果。

第3-6行为执行命令"date"的时间统计结果，其中第4行"real"为实际时间，第5行"user"为用户CPU时间，第6行"sys"为系统CPU时间。

以上三种时间的显示格式均为MMmNN[.FFF]s。

利用下面的指令

```
time -v ps -aux
```

我们可以获得执行 ps -aux 的结果和所花费的系统资源。如下面所列的资料：

```
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.4 1096 472 ? S Apr19 0:04 init
root 2 0.0 0.0 0 0 ? SW Apr19 0:00 [kflushd]
root 3 0.0 0.0 0 0 ? SW Apr19 0:00 [kpiod]
......
root 24269 0.0 1.0 2692 996 pts/3 R 12:16 0:00 ps -aux
Command being timed: "ps -aux"
User time (seconds): 0.05
System time (seconds): 0.06
Percent of CPU this job got: 68%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.16
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 0
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 238
Minor (reclaiming a frame) page faults: 46
Voluntary context switches: 0
Involuntary context switches: 0
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

Linux setup命令

Linux setup命令设置公用程序，是一个启动图形设置系统的命令。

setup 命令：用来配置X，打印设置，时区设置，系统服务，网络配置，配置，防火墙配置，验证配置，鼠标配置。

语法

```
setup
```

setup是一个设置公用程序，提供图形界面的操作方式。在setup中可设置7类的选项：

- 1.登陆认证方式
- 2.键盘组态设置
- 3.鼠标组态设置
- 4.开机时所要启动的系统服务
- 5.声卡组态设置
- 6.时区设置
- 7.X Windows组态设置

Linux sndconfig命令

Linux sndconfig命令用于设置声卡。

sndconfig为声卡设置程序，支持PnP设置，可自动检测并设置PnP声卡。

语法

```
sndconfig [--help][--noautoconfig][--noprobe]
```

参数：

- --help 显示帮助。
- --noautoconfig 不自动设置PnP的声卡。
- --noprobe 不自动检测PnP声卡。

Linux setenv命令

Linux setenv命令用于查询或显示环境变量。

setenv为tsch中查询或设置环境变量的指令。

语法

```
setenv [变量名称][变量值]
```

实例

显示环境变量

```
setenv
```

设置环境变量

```
# setenv USER lx138
```

Linux chkconfig命令

Linux chkconfig命令用于检查，设置系统的各种服务。

这是Red Hat公司遵循GPL规则所开发的程序，它可查询操作系统在每一个执行等级中会执行哪些系统服务，其中包括各类常驻服务。

语法

```
chkconfig [--add][--del][--list][系统服务] 或 chkconfig [--level <等级代号>][系统服务][on/off
```

参数：

- --add　增加所指定的系统服务，让chkconfig指令得以管理它，并同时在系统启动的叙述文件内增加相关数据。
- --del　删除所指定的系统服务，不再由chkconfig指令管理，并同时在系统启动的叙述文件内删除相关数据。
- --level<等级代号>　指定读系统服务要在哪一个执行等级中开启或关毕。

实例

列出chkconfig所知道的所有命令。

```
# chkconfig -list
```

开启服务。

```
# chkconfig telnet on //开启Telnet服务
# chkconfig -list //列出chkconfig所知道的所有的服务的情况
```

关闭服务

```
# chkconfig telnet off  //关闭Telnet服务
# chkconfig -list //列出chkconfig所知道的所有的服务的情况
```

Linux unset命令

Linux unset命令用于删除变量或函数。

unset为shell内建指令，可删除变量或函数。

语法

```
unset [-fv][变量或函数名称]
```

参数：

- -f 　仅删除函数。
- -v 　仅删除变量。

实例

删除环境变量

```
[root@w3cschool.cc ~]# lx="ls -lh" //设定环境变量
[root@w3cschool.cc ~]# $lx //使用环境变量
总用量 116K
-rw-r--r-- 1 root root 2.1K 2008-03-30 anaconda-ks.cfg
drwx------ 3 root root 4.0K 3月 30 21:22 Desktop
-rw-r--r-- 1 root root 50K 2008-03-30 install.log
-rw-r--r-- 1 root root 32K 2008-03-30 install.log.syslog
lrwxrwxrwx 1 root root  9 2008-03-30 qte -> /opt/qte/
[root@w3cschool.cc ~]# set //查看当前的环境变量
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
……省略部分内容
PROMPT_COMMAND='echo -ne "33]0;${USER}@${HOSTNAME%%.*}:${PWD/#$HOME/~}07"'
PS1='[u@h W]$ '
PS2='> '
PS4='+ '
PWD=/root
QTDIR=/usr/lib/qt-3.3
SHELL=/bin/bash
SSH_TTY=/dev/pts/4
SUPPORTED=zh_CN.UTF-8:zh_CN:zh:en_US.UTF-8:en_US:en
SYSFONT=latarcyrheb-sun16
TERM=xterm
UID=0
USER=root
_=-lh
lx='ls -lh'
[root@w3cschool.cc ~]# unset lx //删除环境变量
[root@w3cschool.cc ~]# set //显示当前环境变量
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
……省略部分内容
PROMPT_COMMAND='echo -ne "33]0;${USER}@${HOSTNAME%%.*}:${PWD/#$HOME/~}07"'
PS1='[u@h W]$ '
PS2='> '
PS4='+ '
PWD=/root
QTDIR=/usr/lib/qt-3.3
SHELL=/bin/bash
SSH_TTY=/dev/pts/4
SUPPORTED=zh_CN.UTF-8:zh_CN:zh:en_US.UTF-8:en_US:en
SYSFONT=latarcyrheb-sun16
TERM=xterm
UID=0
USER=root
_=-lh
```

Linux ulimit命令

Linux ulimit命令用于控制shell程序的资源。

ulimit为shell内建指令，可用来控制shell执行程序的资源。

语法

```
ulimit [-aHS][-c <core文件上限>][-d <数据节区大小>][-f <文件大小>][-m <内存大小>][-n <文件数目>]
```

参数：

- -a 　显示目前资源限制的设定。
- -c <core文件上限> 　设定core文件的最大值，单位为区块。
- -d <数据节区大小> 　程序数据节区的最大值，单位为KB。
- -f <文件大小> 　shell所能建立的最大文件，单位为区块。
- -H 　设定资源的硬性限制，也就是管理员所设下的限制。
- -m <内存大小> 　指定可使用内存的上限，单位为KB。
- -n <文件数目> 　指定同一时间最多可开启的文件数。
- -p <缓冲区大小> 　指定管道缓冲区的大小，单位512字节。
- -s <堆叠大小> 　指定堆叠的上限，单位为KB。
- -S 　设定资源的弹性限制。
- -t <CPU时间> 　指定CPU使用时间的上限，单位为秒。
- -u <程序数目> 　用户最多可开启的程序数目。
- -v <虚拟内存大小> 　指定可使用的虚拟内存上限，单位为KB。

实例

显示系统资源的设置

```
[root@w3cschool.cc ~]# ulimit -a
core file size     (blocks, -c) 0
data seg size      (kbytes, -d) unlimited
file size        (blocks, -f) unlimited
pending signals         (-i) 1024
max locked memory    (kbytes, -l) 32
max memory size     (kbytes, -m) unlimited
open files           (-n) 1024
pipe size      (512 bytes, -p) 8
POSIX message queues   (bytes, -q) 819200
stack size       (kbytes, -s) 10240
cpu time        (seconds, -t) unlimited
max user processes       (-u) 4096
virtual memory     (kbytes, -v) unlimited
file locks           (-x) unlimited
[root@w3cschool.cc ~]#
```

设置单一用户程序数目上限

```
[root@w3cschool.cc ~]# ulimit -u 500 //设置单一用户程序上限
[root@w3cschool.cc ~]# ulimit -a
core file size     (blocks, -c) 0
data seg size      (kbytes, -d) unlimited
file size        (blocks, -f) unlimited
pending signals         (-i) 1024
max locked memory    (kbytes, -l) 32
max memory size     (kbytes, -m) unlimited
open files           (-n) 1024
pipe size      (512 bytes, -p) 8
POSIX message queues   (bytes, -q) 819200
stack size       (kbytes, -s) 10240
cpu time        (seconds, -t) unlimited
max user processes       (-u) 500
virtual memory     (kbytes, -v) unlimited
file locks           (-x) unlimited
[root@w3cschool.cc ~]#
```

Linux timeconfig命令

Linux timeconfig命令用于设置时区。

这是Red Hat公司遵循GPL规则所开发的程序，它具有互动式操作界面，您可以轻易地利用方向键和空格键等，设置系统时间所属的时区。

语法

```
timeconfig [--arc][--back][--test][--utc][时区名称]
```

参数：

- --arc 　使用Alpha硬件结构的格式存储系统时间。
- --back 　在互动式界面里，显示Back钮而非Cancel钮。
- --test 　仅作测试，并不真的改变系统的时区。
- --utc 　把硬件时钟上的时间视为CUT，有时也称为UTC或UCT。

实例

```
# timeconfig //设置时区
```

Linux setconsole命令

Linux setconsole命令用于设置系统终端。

setconsole可用来指定系统终端。

语法

```
setconsole [serial][ttya][ttyb]
```

参数：

- serial 　使用PROM终端。
- ttya,cua0或ttyS0 　使用第１个串口设备作为终端。
- ttyb,cua1或ttyS1 　使用第２个串口设备作为终端。
- video 　使用主机上的现卡作为终端。

实例

设置终端

```
# setconsole ttyS0
```

Linux mkkickstart命令

Linux mkkickstart命令用于建立安装的组态文件。

mkkickstart可根据目前系统的设置来建立组态文件，供其他电脑在安装时使用。组态文件的内容包括使用语言，网络环境，系统磁盘状态，以及X Windows的设置等信息。

语法

```
mkkickstart [--bootp][--dhcp][--nonet][--nox][--version][--nfs <远端电脑:路径>]
```

参数：

- --bootp 安装与开机时，使用BOOTP。
- --dhcp 安装与开机时，使用DHCP。
- --nfs<远端电脑:路径> 使用指定的网络路径安装。
- --nonet 不要进行网络设置，即假设在没有网络环境的状态下。
- --nox 不要进行X Windows的环境设置。
- --version 显示版本信息。

实例

构建一个安装组态文件：

```
# mkkickstart --nonet -bootp
```

Linux hwclock命令

Linux hwclock命令用于显示与设定硬件时钟。

在Linux中有硬件时钟与系统时钟等两种时钟。硬件时钟是指主机板上的时钟设备，也就是通常可在BIOS画面设定的时钟。系统时钟则是指kernel中的时钟。当Linux启动时，系统时钟会去读取硬件时钟的设定，之后系统时钟即独立运作。所有Linux相关指令与函数都是读取系统时钟的设定。

语法

```
hwclock [--adjust][--debug][--directisa][--hctosys][--show][--systohc][--test]
[--utc][--version][--set --date=<日期与时间>]
```

参数：

- --adjust　hwclock每次更改硬件时钟时，都会记录在/etc/adjtime文件中。使用--adjust参数，可使hwclock根据先前的记录来估算硬件时钟的偏差，并用来校正目前的硬件时钟。
- --debug　显示hwclock执行时详细的信息。
- --directisa　hwclock预设从/dev/rtc设备来存取硬件时钟。若无法存取时，可用此参数直接以I/O指令来存取硬件时钟。
- --hctosys　将系统时钟调整为与目前的硬件时钟一致。
- --set --date=<日期与时间>　设定硬件时钟。
- --show　显示硬件时钟的时间与日期。
- --systohc　将硬件时钟调整为与目前的系统时钟一致。
- --test　仅测试程序，而不会实际更改硬件时钟。
- --utc　若要使用格林威治时间，请加入此参数，hwclock会执行转换的工作。
- --version　显示版本信息。

实例

显示当前时间

```
# hwclock
2010年05月27日 星期四 18时04分31秒 -0.704214 seconds
```

查看版本信息

```
# hwclock -v
hwclock from util-linux-2.12a
```

Linux apmd命令

Linux apmd命令用于进阶电源管理服务程序。

apmd负责BIOS进阶电源管理(APM)相关的记录，警告与管理工作。

语法

```
apmd [-u v V W][-p <百分比变化量>][-w <百分比值>]
```

参数：

- -p<百分比变化量>或--percentage<百分比变化量> 当电力变化的幅度超出设置的百分比变化量，即记录事件百分比变化量的预设值为5，若设置值超过100，则关闭此功能。
- -u或--utc 将BIOS时钟设为UTC，以便从悬待模式恢复时，将-u参数传送至clock或hwclock程序。
- -v或--verbose 记录所有的APM事件。
- -V或--version 显示版本信息。
- -w<百分比值>或--warn<百分比值> 当电池不在充电状态时，且电池电量低于设置的百分比值，则在syslog(2)的ALERT层记录警告信息。百分比值的预设置为10，若设置为0，则关闭此功能。
- -W或--wall 发出警告信息给所有人。

实例

记录所有的电源管理事件

```
# apmd -v
```

设置BIOS时钟

```
# apmd -utc //设置BIOS时钟为UTC
```

Linux fbset命令

Linux fbset命令用于设置景框缓冲区。

fbset指令可用于设置景框缓冲区的大小，还能调整画面之分辨率，位置，高低宽窄，色彩 深度，并可决定是否启动先卡之各项硬件特性。

语法

```
fbset [-ahinsvVx][-db <信息文件>][-fb <外围设备代号>][--test][显示模式]
```

参数：

- -a或--all 改变所有使用该设备之虚拟终端机的显示模式。
- -db<信息文件> 指定显示模式的信息文件，预设值文件名称为fb.modes，存放在/etc目录下
- -fb<外围设备代号> 指定用来做为输出景框缓冲区之外围设备，预设置为"/dev/fd0"。
- -h或-help 在线帮助。
- -i或--info 列出所有景框缓冲区之相关信息。
- -ifb<外围设备代号> 使用另一个景框缓冲区外围设备之设置值。
- -n或--now 马上改变显示模式。
- -ofb<外围设备代号> 此参数效果和指定"-fb"参数相同。
- -s或--show 列出目前显示模式之设置。
- -v或--verbose 显示指令执行过程。
- -V或--version 显示版本信息。
- -x或--xfree86 使用XFree86兼容模式。
- --test 仅做测试，并不改变现行的显示模式。

实例

设置画面分辨率 和桌面分辨率

```
# fbset -g 800 688 1024 768//画面分辨率为800*600 桌面分辨率为1024*768
```

启动硬件文本加速

```
# fbset -accel true  // 启动硬件文本加速
```

启动广播功能

```
# fbset -bcast true //启动广播功能
```

Linux unalias命令

Linux unalias命令用于删除别名。

unalias为shell内建指令，可删除别名设置。

语法

```
unalias [-a][别名]
```

参数：

- -a 删除全部的别名。

实例

给命令设置别名

```
[root@w3cschool.cc ~]# alias lx=ls
[root@w3cschool.cc ~]# lx
anaconda-ks.cfg Desktop install.log install.log.syslog qte
```

删除别名

```
[root@w3cschool.cc ~]# alias lx //显示别名
alias lx='ls'
[root@w3cschool.cc ~]# unalias lx //删除别名
[root@w3cschool.cc ~]# lx
-bash: lx: command not found
```

Linux SVGATextMode命令

Linux SVGATextMode命令用于加强文字模式的显示画面。

SVGATextMode可用来设置文字模式下的显示画面，包括分辨率，字体和更新频率等。

语法

```
SVGATextMode [-acdfhmnrsv][-t <配置文件>][模式]
```

参数：

- -a 如果新显示模式的屏幕大小与原先不同时，SVGATextMode会执行必要的系统设置。
- -c 维持原有的VGA时脉。
- -d 执行时会显示详细的信息，供排错时参考。
- -f 不要执行配置文件中有关字体载入的指令。
- -h 显示帮助。
- -m 允许1x1的方式来重设屏幕大小。
- -n 仅测试指定的模式。
- -r 通知或重设与屏幕大小相关的程序。
- -s 显示配置文件中所有可用的模式。
- -t<配置文件> 指定配置文件。
- -v SVGATextMode在配置新的显示模式时，预设会先检查垂直与水平的更新更新频率是否在配置文件所指定的范围内，如果不在范围内，则不设置新的显示模式。
- 模式］ [模式]参数必须是配置文件中模式的名称。

Linux命令大全 - 备份压缩

ar	bunzip2	bzip2	bzip2recover
gunzip	unarj	compress	cpio
dump	uuencode	gzexe	gzip
lha	restore	tar	uudecode
unzip	zip	zipinfo	

Linux bzip2recover命令

Linux bzip2recover命令用来修复损坏的.bz2文件。

bzip2是以区块的方式来压缩文件，每个区块视为独立的单位。因此，当某一区块损坏时，便可利用bzip2recover，试着将文件中的区块隔开来，以便解压缩正常的区块。通常只适用在压缩文件很大的情况。

语法

```
bzip2recover [.bz2 压缩文件]
```

实例

修复.bz2文件

bzip2recover col.bz2

Linux bzip2命令

Linux bzip2命令是.bz2文件的压缩程序。

bzip2采用新的压缩演算法，压缩效果比传统的LZ77/LZ78压缩演算法来得好。若没有加上任何参数，bzip2压缩完文件后会产生.bz2的压缩文件，并删除原始的文件。

语法

```
bzip2 [-cdfhkLstvVz][--repetitive-best][--repetitive-fast][- 压缩等级][要压缩的文件]
```

参数：

- -c或--stdout 将压缩与解压缩的结果送到标准输出。
- -d或--decompress 执行解压缩。
- -f或--force bzip2在压缩或解压缩时，若输出文件与现有文件同名，预设不会覆盖现有文件。若要覆盖，请使用此参数。
- -h或--help 显示帮助。
- -k或--keep bzip2在压缩或解压缩后，会删除原始的文件。若要保留原始文件，请使用此参数。
- -s或--small 降低程序执行时内存的使用量。
- -t或--test 测试.bz2压缩文件的完整性。
- -v或--verbose 压缩或解压缩文件时，显示详细的信息。
- -z或--compress 强制执行压缩。
- -L,--license,
- -V或--version 显示版本信息。
- --repetitive-best 若文件中有重复出现的资料时，可利用此参数提高压缩效果。
- --repetitive-fast 若文件中有重复出现的资料时，可利用此参数加快执行速度。
- -压缩等级 压缩时的区块大小。

实例

解压.bz2文件

```
[root@w3cschool.cc ~]# bzip2 -v temp.bz2 //解压文件显示详细处理信息
```

压缩文件

```
[root@w3cschool.cc ~]# bzip2 -c a.c b.c c.c
```

检查文件完整性

```
[root@w3cschool.cc ~]# bzip2 -t temp.bz2
```

Linux bunzip2命令

Linux bunzip2命令是.bz2文件的解压缩程序。

bunzip2可解压缩.bz2格式的压缩文件。bunzip2实际上是bzip2的符号连接，执行bunzip2与bzip2 -d的效果相同。

语法：bunzip2 [-fkLsvV][.bz2压缩文件]

参数：

- -f或--force 　解压缩时，若输出的文件与现有文件同名时，预设不会覆盖现有的文件。若要覆盖，请使用此参数。
- -k或--keep 　在解压缩后，预设会删除原来的压缩文件。若要保留压缩文件，请使用此参数。
- -s或--small 　降低程序执行时，内存的使用量。
- -v或--verbose 　解压缩文件时，显示详细的信息。
- -l,--license,-V或--version 　显示版本信息。

实例

解压.bz2文件

```
# bunzip2 -v temp.bz2 //解压文件显示详细处理信息
```

Linux ar命令

Linux ar命令用于建立或修改备存文件，或是从备存文件中抽取文件。

ar可让您集合许多文件，成为单一的备存文件。在备存文件中，所有成员文件皆保有原来的属性与权限。

语法

```
ar[-dmpqrtx][cfosSuvV][a<成员文件>][b<成员文件>][i<成员文件>][备存文件][成员文件]
```

参数：

必要参数：

- -d 删除备存文件中的成员文件。
- -m 变更成员文件在备存文件中的次序。
- -p 显示备存文件中的成员文件内容。
- -q 将问家附加在备存文件末端。
- -r 将文件插入备存文件中。
- -t 显示备存文件中所包含的文件。
- -x 自备存文件中取出成员文件。

选项参数：

- a<成员文件> 将文件插入备存文件中指定的成员文件之后。
- b<成员文件> 将文件插入备存文件中指定的成员文件之前。
- c 建立备存文件。
- f 为避免过长的文件名不兼容于其他系统的ar指令指令，因此可利用此参数，截掉要放入备存文件中过长的成员文件名称。
- i<成员文件> 将问家插入备存文件中指定的成员文件之前。
- o 保留备存文件中文件的日期。
- s 若备存文件中包含了对象模式，可利用此参数建立备存文件的符号表。
- S 不产生符号表。
- u 只将日期较新文件插入备存文件中。
- v 程序执行时显示详细的信息。
- V 显示版本信息。

实例

打包文件

```
[root@w3cschool.cc ~]# ls   //显示当前目录文件
a.c    b.c d.c   install.log      qte
anaconda-ks.cfg c.c Desktop

[root@w3cschool.cc ~]# ar rv one.bak a.c b.c //打包 a.c b.c文件
ar: 正在创建 one.bak
a - a.c
a - b.c
[root@w3cschool.cc ~]#
```

打包多个文件

```
[root@w3cschool.cc ~]# ar rv two.bak *.c  //打包以.c结尾的文件
ar: 正在创建 two.bak
a - a.c
a - b.c
a - c.c
a - d.c
[root@w3cschool.cc ~]#
```

显示打包文件的内容

```
[root@w3cschool.cc ~]# ar t two.bak
a.c
b.c
c.c
d.c
[root@w3cschool.cc ~]#
```

删除打包文件的成员文件

```
[root@w3cschool.cc ~]# ar d two.bak a.c b.c c.c
[root@w3cschool.cc ~]# ar t two.bak
d.c
```

Linux gunzip命令

Linux gunzip命令用于解压文件。

gunzip是个使用广泛的解压缩程序，它用于解开被gzip压缩过的文件，这些压缩文件预设最后的扩展名为".gz"。事实上gunzip就是gzip的硬连接，因此不论是压缩或解压缩，都可通过gzip指令单独完成。

语法

参数：

```
gunzip [-acfhlLnNqrtvV][-s <压缩字尾字符串>][文件...] 或 gunzip [-acfhlLnNqrtvV][-s <压缩字尾
```

- -a或--ascii　使用ASCII文字模式。
- -c或--stdout或--to-stdout　把解压后的文件输出到标准输出设备。
- -f或-force　强行解开压缩文件，不理会文件名称或硬连接是否存在以及该文件是否为符号连接。
- -h或--help　在线帮助。
- -l或--list　列出压缩文件的相关信息。
- -L或--license　显示版本与版权信息。
- -n或--no-name　解压缩时，若压缩文件内含有远来的文件名称及时间戳记，则将其忽略不予处理。
- -N或--name　解压缩时，若压缩文件内含有原来的文件名称及时间戳记，则将其回存到解开的文件上。
- -q或--quiet　不显示警告信息。
- -r或--recursive　递归处理，将指定目录下的所有文件及子目录一并处理。
- -S<压缩字尾字符串>或--suffix<压缩字尾字符串>　更改压缩字尾字符串。
- -t或--test　测试压缩文件是否正确无误。
- -v或--verbose　显示指令执行过程。
- -V或--version 显示版本信息。

实例

```
<p>解压文件
</p>
<pre>
# gunzip ab.gz
```

Linux unarj命令

Linux unarj命令用于解压缩.arj文件。

unarj为.arj压缩文件的压缩程序。

语法

```
unarj [eltx][.arj压缩文件]
```

参数：

- e 解压缩.arj文件。
- l 显示压缩文件内所包含的文件。
- t 检查压缩文件是否正确。
- x 解压缩时保留原有的路径。

实例

解压.arj文件

```
# unarj e test.arj
```

Linux compress命令

Linux compress命令是一个相当古老的 unix 档案压缩指令，压缩后的档案会加上一个 .Z 延伸档名以区别未压缩的档案，压缩后的档案可以以 uncompress 解压。若要将数个档案压成一个压缩档，必须先将档案 tar 起来再压缩。由于 gzip 可以产生更理想的压缩比例，一般人多已改用 gzip 为档案压缩工具。

语法

```
compress [-dfvcV] [-b maxbits] [file ...]
```

参数：

- c 输出结果至标准输出设备（一般指荧幕）
- f 强迫写入档案，若目的档已经存在，则会被覆盖 (force)
- v 将程序执行的讯息印在荧幕上 (verbose)
- b 设定共同字串数的上限，以位元计算，可以设定的值为 9 至 16 bits 。由于值越大，能使用的共同字串就 越多，压缩比例就越大，所以一般使用预设值 16 bits (bits)
- d 将压缩档解压缩
- V 列出版本讯息
- 范例：
- 将 source.dat 压缩成 source.dat.Z ，若 source.dat.Z 已经存在，内容则会被压缩档覆盖。
- compress -f source.dat
- 将 source.dat 压缩成 source.dat.Z ，并列印出压缩比例。
- -v 与 -f 可以一起使用
- compress -vf source.dat
- 将压缩后的资料输出后再导入 target.dat.Z 可以改变压缩档名。
- compress -c source.dat > target.dat.Z
- -b 的值越大，压缩比例就越大，范围是 9-16 ，预设值是 16 。
- compress -b 12 source.dat
- 将 source.dat.Z 解压成 source.dat ，若档案已经存在，使用者按 y 以确定覆盖档案，若使用 -df 程序则会自动覆盖档案。由于系统会自动加入 .Z 为延伸档名，所以 source.dat 会自动当作 source.dat.Z 处理。
- compress -d source.dat
- compress -d source.dat.Z

压缩文件

```
[root@w3cschool.cc ~]# compress abc.h
[root@w3cschool.cc ~]# ls

abc.h.Z
```

解压文件

```
[root@w3cschool.cc ~]# compress -d abc.h.Z
[root@w3cschool.cc ~]# ls

abc.h.
```

按指定压缩比例进行压缩

```
[root@w3cschool.cc ~]# compress -b 7 abc.h
```

强制压缩文件夹

```
[root@w3cschool.cc ~]# compress -rf /home/abc/
```

Linux cpio命令

Linux cpio命令用于备份文件。

cpio是用来建立，还原备份档的工具程序，它可以加入，解开cpio或tra备份档内的文件。

语法

```
cpio [-0aABckLovV][-C <输入/输出大小>][-F <备份档>][-H <备份格式>][-O <备份档>][--block-size=<
```

参数：

- -0或--null 　接受新增列控制字符，通常配合find指令的"-print0"参数使用。
- -a或--reset-access-time 　重新设置文件的存取时间。
- -A或--append 　附加到已存在的备份档中，且这个备份档必须存放在磁盘上，而不能放置于磁带机里。
- -b或--swap 　此参数的效果和同时指定"-sS"参数相同。
- -B 　将输入/输出的区块大小改成5210 Bytes。
- -c 　使用旧ASCII备份格式。
- -C<区块大小>或--io-size=<区块大小> 　设置输入/输出的区块大小，单位是Byte。
- -d或--make-directories 　如有需要cpio会自行建立目录。
- -E<范本文件>或--pattern-file=<范本文件> 　指定范本文件，其内含有一个或多个范本样式，让cpio解开符合范本条件的文件，格式为每列一个范本样式。
- -f或--nonmatching 　让cpio解开所有不符合范本条件的文件。
- -F<备份档>或--file=<备份档> 　指定备份档的名称，用来取代标准输入或输出，也能借此通过网络使用另一台主机的保存设备存取备份档。
- -H<备份格式> 　指定备份时欲使用的文件格式。
- -i或--extract 　执行copy-in模式，还原备份档。
- -I<备份档> 　指定备份档的名称，用来取代标准输入，也能借此通过网络使用另一台主机的保存设备读取备份档。
- -k 　此参数将忽略不予处理，仅负责解决cpio不同版本间的兼容性问题。
- -l或--link 　以硬连接的方式取代复制文件，可在copy-pass模式下运用。
- -L或--dereference 　不建立符号连接，直接复制该连接所指向的原始文件。
- -m或preserve-modification-time 　不去更换文件的更改时间。
- -M<回传信息>或--message=<回传信息> 　设置更换保存媒体的信息。
- -n或--numeric-uid-gid 　使用"-tv"参数列出备份档的内容时，若再加上参数"-n"，则会以用户识别码和群组识别码替代拥有者和群组名称列出文件清单。
- -o或--create 　执行copy-out模式，建立备份档。

- -O<备份档> 指定备份档的名称，用来取代标准输出，也能借此通过网络 使用另一台主机的保存设备存放备份档。
- -p或--pass-through 执行copy-pass模式，略过备份步骤，直接将文件复制到目的目录。
- -r或--rename 当有文件名称需要更动时，采用互动模式。
- -R<拥有者><:/.><所属群组>或
- ----owner<拥有者><:/.><所属群组> 在copy-in模式还原备份档，或copy-pass模式复制文件时，可指定这些备份，复制的文件的拥有者与所属群组。
- -s或--swap-bytes 交换每对字节的内容。
- -S或--swap-halfwords 交换每半个字节的内容。
- -t或--list 将输入的内容呈现出来。
- -u或--unconditional 置换所有文件，不论日期时间的新旧与否，皆不予询问而直接覆盖。
- -v或--verbose 详细显示指令的执行过程。
- -V或--dot 执行指令时，在每个文件的执行程序前面加上"."号
- --block-size=<区块大小> 设置输入/输出的区块大小，假如设置数值为5，则区块大小为2500，若设置成10，则区块大小为5120，依次类推。
- --force-local 强制将备份档存放在本地主机。
- --help 在线帮助。
- --no-absolute-filenames 使用相对路径建立文件名称。
- --no-preserve-owner 不保留文件的拥有者，谁解开了备份档，那些文件就归谁所有。
- -only-verify-crc 当备份档采用CRC备份格式时，可使用这项参数检查备份档内的每个文件是否正确无误。
- --quiet 不显示复制了多少区块。
- --sparse 倘若一个文件内含大量的连续0字节，则将此文件存成稀疏文件。
- --version 显示版本信息。

实例

制作备份文件

```
[root@w3cschool.cc var]# ll //显示当前目录下的文件
总用量 164
drwxr-xr-x  2 root  root  4096 2008-03-30 account
drwxr-xr-x  9 root  root  4096 2008-03-30 cache
drwxr-xr-x  3 netdump netdump 4096 2008-03-30 crash
drwxr-xr-x  3 root  root  4096 2008-03-30 db
drwxr-xr-x  3 root  root  4096 2008-03-30 empty
drwxr-xr-x  3 root  root  4096 2008-03-30 ftp
drwxrwx--T  2 root  gdm   4096 4月 9 20:17 gdm
drwxr-xr-x 25 root  root  4096 2008-03-30 lib
drwxr-xr-x  2 root  root  4096 2004-08-13 local
drwxrwxr-x  6 root  lock  4096 5月 8 15:25 lock
drwxr-xr-x 14 root  root  4096 5月 8 15:14 log
lrwxrwxrwx  1 root  root   10 2008-03-30 mail -> spool/mail
drwxr-xr-x  2 root  root  4096 2004-08-13 nis
drwxr-xr-x  2 root  root  4096 2004-08-13 opt
drwxr-xr-x  2 root  root  4096 2004-08-13 preserve
drwxr-xr-x 16 root  root  4096 5月 8 15:14 run
drwxr-xr-x 16 root  root  4096 2008-03-30 spool
drwxrwxrwt  3 root  root  4096 1月 13 18:53 tmp
drwx------  2 root  root  4096 2004-07-08 tux
drwxr-xr-x  8 root  root  4096 1月 19 19:39 www
drwxr-xr-x  3 root  root  4096 2008-03-30 yp
[root@w3cschool.cc var]# ls | cpio -o >123.cpio //制作备份文件
25 blocks
[root@w3cschool.cc var]# ll //显示当前目录下的文件
总用量 172
-rw-r--r--  1 root  root  1024 5月 24 13:06 123.cpio
drwxr-xr-x  2 root  root  4096 2008-03-30 account
drwxr-xr-x  9 root  root  4096 2008-03-30 cache
drwxr-xr-x  3 netdump netdump 4096 2008-03-30 crash
drwxr-xr-x  3 root  root  4096 2008-03-30 db
drwxr-xr-x  3 root  root  4096 2008-03-30 empty
drwxr-xr-x  3 root  root  4096 2008-03-30 ftp
drwxrwx--T  2 root  gdm   4096 4月 9 20:17 gdm
drwxr-xr-x 25 root  root  4096 2008-03-30 lib
drwxr-xr-x  2 root  root  4096 2004-08-13 local
drwxrwxr-x  6 root  lock  4096 5月 8 15:25 lock
drwxr-xr-x 14 root  root  4096 5月 8 15:14 log
lrwxrwxrwx  1 root  root   10 2008-03-30 mail -> spool/mail
drwxr-xr-x  2 root  root  4096 2004-08-13 nis
drwxr-xr-x  2 root  root  4096 2004-08-13 opt
drwxr-xr-x  2 root  root  4096 2004-08-13 preserve
drwxr-xr-x 16 root  root  4096 5月 8 15:14 run
drwxr-xr-x 16 root  root  4096 2008-03-30 spool
drwxrwxrwt  3 root  root  4096 1月 13 18:53 tmp
drwx------  2 root  root  4096 2004-07-08 tux
drwxr-xr-x  8 root  root  4096 1月 19 19:39 www
drwxr-xr-x  3 root  root  4096 2008-03-30 yp
[root@w3cschool.cc var]#
```

解压备份文件

```
[root@w3cschool.cc var]# ls | cpio -i -l 123.cpio
```

解压缩备份文件，并列出详细信息

```
[root@w3cschool.cc var]# cpio -t -I 123.cpio
123.cpio
a.c
b.c
c.c
……省略部分结果
```

强制解压缩

```
[root@w3cschool.cc var]# cpio -i -u -I 123.cpio
```

解压缩时进行反向匹配，指定不解压的文件

```
[root@w3cschool.cc var]# cpio -i -I 123.cpio -f *.c

//不解压.c结尾的文件
```

向指定的.cpio文件添加文件

```
[root@w3cschool.cc var]# ls
123.cpio crash ftp local mail preserve tmp yp
account  db    gdm lock  nis  run       tux
cache     empty lib log  opt  spool   www
[root@w3cschool.cc var]# cpio -o -O 123.cpio -A
db  //用户输入 按下Ctrl+D结束输入
1 block
[root@w3cschool.cc var]#
```

从标准输入备份文件

```
[root@w3cschool.cc test]# ls
a. a.c b.c c.c d.c    f.c
[root@w3cschool.cc test]# cpio -o >123.cpio
a.c  //用户输入
b.c
c.c  //按下Ctrl+D完成输入
3 block
[root@w3cschool.cc test]#
```

复制文件

```
[root@w3cschool.cc test]# cpio -p /root
a.c  //用户输入
b.c
c.c  //按下Ctrl+D完成输入
3 block
```

Linux dump命令

Linux dump命令用于备份文件系统。

dump为备份工具程序，可将目录或整个文件系统备份至指定的设备，或备份成一个大文件。

语法

```
dump [-cnu][-0123456789][-b <区块大小>][-B <区块数目>][-d <密度>][-f <设备名称>][-h <层级>][-s
```

参数：

- -0123456789　备份的层级。
- -b<区块大小>　指定区块的大小，单位为KB。
- -B<区块数目>　指定备份卷册的区块数目。
- -c　修改备份磁带预设的密度与容量。
- -d<密度>　设置磁带的密度。单位为BPI。
- -f<设备名称>　指定备份设备。
- -h<层级>　当备份层级等于或大雨指定的层级时，将不备份用户标示为"nodump"的文件。
- -n　当备份工作需要管理员介入时，向所有"operator"群组中的使用者发出通知。
- -s<磁带长度>　备份磁带的长度，单位为英尺。
- -T<日期>　指定开始备份的时间与日期。
- -u　备份完毕后，在/etc/dumpdates中记录备份的文件系统，层级，日期与时间等。
- -w　与-W类似，但仅显示需要备份的文件。
- -W　显示需要备份的文件及其最后一次备份的层级，时间与日期。

实例

备份文件到磁带

```
# dump -0 -u /dev/tape /home/
```

其中"-0"参数指定的是备份等级"-u"要求备份完毕之后将相应的信息存储到文件 /etc/dumpdates 留作记录

Linux uuencode命令

Linux uuencode命令用于将uuencode编码后的档案还原。

早期在许多 unix 系统的传送协定只能传送七位元字元，并不支援二进位档案，像中文文字档就有用到八位元，所以无法完整地送到另一架机器上。 uuencode 指令，可以将二进位档转换成七位元的档案，传送到另一架机器上再以 uudecode 还原。最常见的是用在以电子邮件传送二进位档。uuencode 编码后的资料都以 begin 开始，以 end 作为结束。

语法

```
compress[必要参数][选择参数][目录或者文件]
```

参数说明：

必要参数：

- 无

选择参数：

- h 显示帮助信息
- v 显示版本信息

实例

还原档案

```
# uuencode test.uud
```

Linux restore命令

Linux restore命令用来还原由dump操作所备份下来的文件或整个文件系统(一个分区)。

restore 指令所进行的操作和dump指令相反，dump操作可用来备份文件，而restore操作则是写回这些已备份的文件。

语法

```
restore [-cCvy][-b <区块大小>][-D <文件系统>][-f <备份文件>][-s <文件编号>] 或 restore [-chimvy
```

参数：

- -b<区块大小> 设置区块大小，单位是Byte。
- -c 不检查dump操作的备份格式，仅准许读取使用旧格式的备份文件。
- -C 使用对比模式，将备份的文件与现行的文件相互对比。
- -D<文件系统> 允许用户指定文件系统的名称。
- -f<备份文件> 从指定的文件中读取备份数据，进行还原操作。
- -h 仅解出目录而不包括与该目录相关的所有文件。
- -i 使用互动模式，在进行还原操作时，restore指令将依序询问用户。
- -m 解开符合指定的inode编号的文件或目录而非采用文件名称指定。
- -r 进行还原操作。
- -R 全面还原文件系统时，检查应从何处开始进行。
- -s<文件编号> 当备份数据超过一卷磁带时，您可以指定备份文件的编号。
- -t 指定文件名称，若该文件已存在备份文件中，则列出它们的名称。
- -v 显示指令执行过程。
- -x 设置文件名称，且从指定的存储媒体里读入它们，若该文件已存在在备份文件中，则将其还原到文件系统内。
- -y 不询问任何问题，一律以同意回答并继续执行指令。

Linux lha命令

Linux lha命令用于压缩或解压缩文件。

lha是从lharc演变而来的压缩程序，文件经它压缩后，会另外产生具有".lzh"扩展名的压缩文件。

语法

```
lha [-acdfglmnpqtuvx][-a <0/1/2>/u</0/1/2>][-<a/c/u>d][-<e/x>i][-<a/u>o][-<e/x>w=<目的目录
```

参数：

- -a或a 压缩文件，并加入到压缩文件内。
- -a<0/1/2>/u</0/1/2> 压缩文件时，采用不同的文件头。
- -c或c 压缩文件，重新建构新的压缩文件后，再将其加入。
- -d或d 从压缩文件内删除指定的文件。
- -<a/c/u>d或<a/c/u>d 压缩文件，然后将其加入，重新建构，更新压缩文件或，删除原始文件，也就是把文件移到压缩文件中。
- -e或e 解开压缩文件。
- -f或f 强制执行lha命令，在解压时会直接覆盖已有的文件而不加以询问。
- -g或g 使用通用的压缩格式，便于解决兼容性的问题。
- -<e/x>i或<e/x>i 解开压缩文件时，忽略保存在压缩文件内的文件路径，直接将其解压后存放在现行目录下或是指定的目录中。
- -l或l 列出压缩文件的相关信息。
- -m或m 此参数的效果和同时指定"-ad"参数相同。
- -n或n 不执行指令，仅列出实际执行会进行的动作。
- -<a/u>o或<a/u>o 采用lharc兼容格式，将压缩后的文件加入，更新压缩文件。
- -p或p 从压缩文件内输出到标准输出设备。
- -q或q 不显示指令执行过程。
- -t或t 检查备份文件内的每个文件是否正确无误。
- -u或u 更换较新的文件到压缩文件内。
- -u</0/1/2>或u</0/1/2> 在文件压缩时采用不同的文件头，然后更新到压缩文件内。
- -v或v 详细列出压缩文件的相关信息。
- -<e/x>w=<目的目录>或<e/x>w=<目的目录> 指定解压缩的目录。
- -x或x 解开压缩文件。
- -<a/u>z或<a/u>z 不压缩文件，直接把它加入，更新压缩文件。

实例

缩文件

```
# lha -a abc.lhz a.b //压缩a.b文件，压缩后生成 abc.lhz文件
```

压缩目录

```
# lha -a abc2 /home/hnlinux
```

解压文件到当前目录

```
# lha -xiw=agis abc  //解压文件abc
```

Linux gzip命令

Linux gzip命令用于压缩文件。

gzip是个使用广泛的压缩程序，文件经它压缩过后，其名称后面会多出".gz"的扩展名。

语法

```
gzip [-acdfhlLnNqrtvV][-S &lt;压缩字尾字符串&gt;][-&lt;压缩效率&gt;][--best/fast][文件...] 或
```

参数：

- -a或--ascii 　使用ASCII文字模式。
- -c或--stdout或--to-stdout 　把压缩后的文件输出到标准输出设备，不去更动原始文件。
- -d或--decompress或----uncompress 　解开压缩文件。
- -f或--force 　强行压缩文件。不理会文件名称或硬连接是否存在以及该文件是否为符号连接。
- -h或--help 　在线帮助。
- -l或--list 　列出压缩文件的相关信息。
- -L或--license 　显示版本与版权信息。
- -n或--no-name 　压缩文件时，不保存原来的文件名称及时间戳记。
- -N或--name 　压缩文件时，保存原来的文件名称及时间戳记。
- -q或--quiet 　不显示警告信息。
- -r或--recursive 　递归处理，将指定目录下的所有文件及子目录一并处理。
- -S<压缩字尾字符串>或----suffix<压缩字尾字符串> 　更改压缩字尾字符串。
- -t或--test 　测试压缩文件是否正确无误。
- -v或--verbose 　显示指令执行过程。
- -V或--version 　显示版本信息。
- -<压缩效率> 　压缩效率是一个介于1－9的数值，预设值为"6"，指定愈大的数值，压缩效率就会愈高。
- --best 　此参数的效果和指定"-9"参数相同。
- --fast 　此参数的效果和指定"-1"参数相同。

实例

压缩文件

```
[root@w3cschool.cc a]# ls //显示当前目录文件
a.c b.h d.cpp
[root@w3cschool.cc a]# gzip * //压缩目录下的所有文件
[root@w3cschool.cc a]# ls //显示当前目录文件
a.c.gz    b.h.gz    d.cpp.gz
[root@w3cschool.cc a]#
```

接范例1， 列出详细的信息

```
[root@w3cschool.cc a]# gzip -dv * //解压文件，并列出详细信息
a.c.gz:     0.0% -- replaced with a.c
b.h.gz:     0.0% -- replaced with b.h
d.cpp.gz:     0.0% -- replaced with d.cpp
[root@w3cschool.cc a]#
```

接范例1，显示压缩文件的信息

```
[root@w3cschool.cc a]# gzip -l *
     compressed    uncompressed ratio uncompressed_name
         24           0  0.0% a.c
         24           0  0.0% b.h
         26           0  0.0% d.cpp
```

Linux gzexe命令

Linux gzexe命令用于压缩执行文件。

gzexe是用来压缩执行文件的程序。当您去执行被压缩过的执行文件时，该文件会自动解压然后继续执行，和使用一般的执行文件相同。

语法

```
gzexe [-d][执行文件...]
```

参数：

- -d   解开压缩文件。

实例

压缩可执行文件

```
# gzexe abc
```

Linux zipinfo命令

Linux zipinfo命令用于列出压缩文件信息。

执行zipinfo指令可得知zip压缩文件的详细信息。

语法

```
zipinfo [-12hlmMstTvz][压缩文件][文件...][-x <范本样式>]
```

参数：

- -1 只列出文件名称。
- -2 此参数的效果和指定"-1"参数类似，但可搭配"-h","-t"和"-z"参数使用。
- -h 只列出压缩文件的文件名称。
- -l 此参数的效果和指定"-m"参数类似，但会列出原始文件的大小而非每个文件的压缩率。
- -m 此参数的效果和指定"-s"参数类似，但多会列出每个文件的压缩率。
- -M 若信息内容超过一个画面，则采用类似more指令的方式列出信息。
- -s 用类似执行"ls -l"指令的效果列出压缩文件内容。
- -t 只列出压缩文件内所包含的文件数目，压缩前后的文件大小及压缩率。
- -T 将压缩文件内每个文件的日期时间用年，月，日，时，分，秒的顺序列出。
- -v 详细显示压缩文件内每一个文件的信息。
- -x<范本样式> 不列出符合条件的文件的信息。
- -z 如果压缩文件内含有注释，就将注释显示出来。

实例

显示压缩文件信息

```
[root@w3cschool.cc a]# zipinfo cp.zip
Archive: cp.zip  486 bytes  4 files
-rw-r--r-- 2.3 unx    0 bx stor 24-May-10 18:54 a.c
-rw-r--r-- 2.3 unx    0 bx stor 24-May-10 18:54 b.c
-rw-r--r-- 2.3 unx    0 bx stor 24-May-10 18:54 c.c
-rw-r--r-- 2.3 unx    0 bx stor 24-May-10 18:54 e.c
4 files, 0 bytes uncompressed, 0 bytes compressed: 0.0%
[root@w3cschool.cc a]#
```

显示压缩文件中每个文件的信息

```
[root@w3cschool.cc a]# zipinfo -v cp.zip
Archive: cp.zip  486 bytes  4 files

End-of-central-directory record:
```

```
-------------------------------

  Actual offset of end-of-central-dir record:     464 (000001D0h)
  Expected offset of end-of-central-dir record:    464 (000001D0h)
  (based on the length of the central directory and its expected offset)

  This zipfile constitutes the sole disk of a single-part archive; its
  central directory contains 4 entries. The central directory is 248
  (000000F8h) bytes long, and its (expected) offset in bytes from the
  beginning of the zipfile is 216 (000000D8h).

  There is no zipfile comment.

Central directory entry #1:
---------------------------

  a.c

  offset of local header from start of archive:   0 (00000000h) bytes
  file system or operating system of origin:    Unix
  version of encoding software:           2.3
  minimum file system compatibility required:    MS-DOS, OS/2 or NT FAT
  minimum software version required to extract:   1.0
  compression method:                none (stored)
  file security status:               not encrypted
  extended local header:              no
  file last modified on (DOS date/time):      2010 May 24 18:54:26
  file last modified on (UT extra field modtime):  2010 May 24 18:54:26 local
  file last modified on (UT extra field modtime):  2010 May 24 10:54:26 UTC
  32-bit CRC value (hex):              00000000
  compressed size:                 0 bytes
  uncompressed size:                0 bytes
  length of filename:                3 characters
  length of extra field:              13 bytes
  length of file comment:              0 characters
  disk number on which file begins:         disk 1
  apparent file type:                binary
  Unix file attributes (100644 octal):       -rw-r--r--
  MS-DOS file attributes (00 hex):         none

  The central-directory extra field contains:
  - A subfield with ID 0x5455 (universal time) and 5 data bytes.
   The local extra field has UTC/GMT modification/access times.
  - A subfield with ID 0x7855 (Unix UID/GID) and 0 data bytes.

  There is no file comment.

Central directory entry #2:
---------------------------

  b.c

  offset of local header from start of archive:   54 (00000036h) bytes
  file system or operating system of origin:    Unix
  version of encoding software:           2.3
  minimum file system compatibility required:    MS-DOS, OS/2 or NT FAT
  minimum software version required to extract:   1.0
  compression method:                none (stored)
  file security status:               not encrypted
  extended local header:              no
  file last modified on (DOS date/time):      2010 May 24 18:54:26
  file last modified on (UT extra field modtime):  2010 May 24 18:54:26 local
  file last modified on (UT extra field modtime):  2010 May 24 10:54:26 UTC
  32-bit CRC value (hex):              00000000
  compressed size:                 0 bytes
  uncompressed size:                0 bytes
  length of filename:                3 characters
  length of extra field:              13 bytes
  length of file comment:              0 characters
  disk number on which file begins:         disk 1
  apparent file type:                binary
  Unix file attributes (100644 octal):       -rw-r--r--
```

```
  MS-DOS file attributes (00 hex):         none

  The central-directory extra field contains:
  - A subfield with ID 0x5455 (universal time) and 5 data bytes.
    The local extra field has UTC/GMT modification/access times.
  - A subfield with ID 0x7855 (Unix UID/GID) and 0 data bytes.

  There is no file comment.

Central directory entry #3:
---------------------------

  c.c

  offset of local header from start of archive:   108 (0000006Ch) bytes
  file system or operating system of origin:    Unix
  version of encoding software:           2.3
  minimum file system compatibility required:    MS-DOS, OS/2 or NT FAT
  minimum software version required to extract:   1.0
  compression method:                none (stored)
  file security status:               not encrypted
  extended local header:              no
  file last modified on (DOS date/time):      2010 May 24 18:54:26
  file last modified on (UT extra field modtime):  2010 May 24 18:54:26 local
  file last modified on (UT extra field modtime):  2010 May 24 10:54:26 UTC
  32-bit CRC value (hex):              00000000
  compressed size:                 0 bytes
  uncompressed size:                0 bytes
  length of filename:                3 characters
  length of extra field:              13 bytes
  length of file comment:              0 characters
  disk number on which file begins:         disk 1
  apparent file type:                binary
  Unix file attributes (100644 octal):       -rw-r--r--
  MS-DOS file attributes (00 hex):         none

  The central-directory extra field contains:
  - A subfield with ID 0x5455 (universal time) and 5 data bytes.
    The local extra field has UTC/GMT modification/access times.
  - A subfield with ID 0x7855 (Unix UID/GID) and 0 data bytes.

  There is no file comment.

Central directory entry #4:
---------------------------

  e.c

  offset of local header from start of archive:   162 (000000A2h) bytes
  file system or operating system of origin:    Unix
  version of encoding software:           2.3
  minimum file system compatibility required:    MS-DOS, OS/2 or NT FAT
  minimum software version required to extract:   1.0
  compression method:                none (stored)
  file security status:               not encrypted
  extended local header:              no
  file last modified on (DOS date/time):      2010 May 24 18:54:26
  file last modified on (UT extra field modtime):  2010 May 24 18:54:26 local
  file last modified on (UT extra field modtime):  2010 May 24 10:54:26 UTC
  32-bit CRC value (hex):              00000000
  compressed size:                 0 bytes
  uncompressed size:                0 bytes
  length of filename:                3 characters
  length of extra field:              13 bytes
  length of file comment:              0 characters
  disk number on which file begins:         disk 1
  apparent file type:                binary
  Unix file attributes (100644 octal):       -rw-r--r--
  MS-DOS file attributes (00 hex):         none

  The central-directory extra field contains:
  - A subfield with ID 0x5455 (universal time) and 5 data bytes.
```

```
    The local extra field has UTC/GMT modification/access times.
  - A subfield with ID 0x7855 (Unix UID/GID) and 0 data bytes.

  There is no file comment.
```

Linux zip命令

Linux zip命令用于压缩文件。

zip是个使用广泛的压缩程序，文件经它压缩后会另外产生具有".zip"扩展名的压缩文件。

语法

```
zip [-AcdDfFghjJKlLmoqrSTuvVwXyz$][-b <工作目录>][-ll][-n <字尾字符串>][-t <日期时间>][-<压缩效
```

参数：

- -A 调整可执行的自动解压缩文件。
- -b<工作目录> 指定暂时存放文件的目录。
- -c 替每个被压缩的文件加上注释。
- -d 从压缩文件内删除指定的文件。
- -D 压缩文件内不建立目录名称。
- -f 此参数的效果和指定"-u"参数类似，但不仅更新既有文件，如果某些文件原本不存在于压缩文件内，使用本参数会一并将其加入压缩文件中。
- -F 尝试修复已损坏的压缩文件。
- -g 将文件压缩后附加在既有的压缩文件之后，而非另行建立新的压缩文件。
- -h 在线帮助。
- -i<范本样式> 只压缩符合条件的文件。
- -j 只保存文件名称及其内容，而不存放任何目录名称。
- -J 删除压缩文件前面不必要的数据。
- -k 使用MS-DOS兼容格式的文件名称。
- -l 压缩文件时，把LF字符置换成LF+CR字符。
- -ll 压缩文件时，把LF+CR字符置换成LF字符。
- -L 显示版权信息。
- -m 将文件压缩并加入压缩文件后，删除原始文件，即把文件移到压缩文件中。
- -n<字尾字符串> 不压缩具有特定字尾字符串的文件。
- -o 以压缩文件内拥有最新更改时间的文件为准，将压缩文件的更改时间设成和该文件相同。
- -q 不显示指令执行过程。
- -r 递归处理，将指定目录下的所有文件和子目录一并处理。
- -S 包含系统和隐藏文件。
- -t<日期时间> 把压缩文件的日期设成指定的日期。
- -T 检查备份文件内的每个文件是否正确无误。

- -u 更换较新的文件到压缩文件内。
- -v 显示指令执行过程或显示版本信息。
- -V 保存VMS操作系统的文件属性。
- -w 在文件名称里假如版本编号，本参数仅在VMS操作系统下有效。
- -x<范本样式> 压缩时排除符合条件的文件。
- -X 不保存额外的文件属性。
- -y 直接保存符号连接，而非该连接所指向的文件，本参数仅在UNIX之类的系统下有效。
- -z 替压缩文件加上注释。
- -$ 保存第一个被压缩文件所在磁盘的卷册名称。
- -<压缩效率> 压缩效率是一个介于1-9的数值。

实例

压缩文件

```
# zip -v cp.zip a.c b.c c.c e.c
 adding: a.c    (in=0) (out=0) (stored 0%)
 adding: b.c    (in=0) (out=0) (stored 0%)
 adding: c.c    (in=0) (out=0) (stored 0%)
 adding: e.c    (in=0) (out=0) (stored 0%)
total bytes=0, compressed=0 -> 0% savings
```

压缩文件

```
# [root@ubuntu a]# zip -v cp2.zip *

#
```

压缩目录

```
# zip -r cp3.zip /root/
```

从压缩文件中删除文件

```
# zip -dv cp.zip a.c
```

Linux unzip命令

Linux unzip命令用于解压缩zip文件

unzip为.zip压缩文件的解压缩程序。

语法

```
unzip [-cflptuvz][-agCjLMnoqsVX][-P <密码>][.zip文件][文件][-d <目录>][-x <文件>] 或 unzip [-
```

参数：

- -c 将解压缩的结果显示到屏幕上，并对字符做适当的转换。
- -f 更新现有的文件。
- -l 显示压缩文件内所包含的文件。
- -p 与-c参数类似，会将解压缩的结果显示到屏幕上，但不会执行任何的转换。
- -t 检查压缩文件是否正确。
- -u 与-f参数类似，但是除了更新现有的文件外，也会将压缩文件中的其他文件解压缩到目录中。
- -v 执行是时显示详细的信息。
- -z 仅显示压缩文件的备注文字。
- -a 对文本文件进行必要的字符转换。
- -b 不要对文本文件进行字符转换。
- -C 压缩文件中的文件名称区分大小写。
- -j 不处理压缩文件中原有的目录路径。
- -L 将压缩文件中的全部文件名改为小写。
- -M 将输出结果送到more程序处理。
- -n 解压缩时不要覆盖原有的文件。
- -o 不必先询问用户，unzip执行后覆盖原有文件。
- -P<密码> 使用zip的密码选项。
- -q 执行时不显示任何信息。
- -s 将文件名中的空白字符转换为底线字符。
- -V 保留VMS的文件版本信息。
- -X 解压缩时同时回存文件原来的UID/GID。
- [.zip文件] 指定.zip压缩文件。
- [文件] 指定要处理.zip压缩文件中的哪些文件。
- -d<目录> 指定文件解压缩后所要存储的目录。
- -x<文件> 指定不要处理.zip压缩文件中的哪些文件。

- -Z unzip -Z等于执行zipinfo指令。

实例

显示压缩文件信息

```
# unzip -l abc.zip
Archive: abc.zip
 Length   Date  Time  Name
--------  ----  ----  ----
  94618 05-21-10 20:44  a11.jpg
  202001 05-21-10 20:44  a22.jpg
    16 05-22-10 15:01  11.txt
  46468 05-23-10 10:30  w456.JPG
  140085 03-14-10 21:49  my.asp
--------          -------
  483188          5 files
```

解压文件

```
# unzip -v abc.zip
Archive: abc.zip
Length  Method  Size Ratio  Date  Time  CRC-32  Name
-------- ------ ------- -----  ----  ----  ------  ----
  94618 Defl:N  93353  1% 05-21-10 20:44 9e661437 a11.jpg
 202001 Defl:N  201833  0% 05-21-10 20:44 1da462eb a22.jpg
   16 Stored    16  0% 05-22-10 15:01 ae8a9910 ? +-|￥+-? (11).txt
  46468 Defl:N  39997 14% 05-23-10 10:30 962861f2 w456.JPG
 140085 Defl:N  36765 74% 03-14-10 21:49 836fcc3f my.asp
--------     ------- ---             -------
 483188      371964 23%              5 files
```

Linux uudecode命令

Linuxuudecode 将 uuencode 编码后的档案还原， uudecode 只会将 begin 与 end 标记之间的编码资料还原，程序会跳过标记以外的资料。

语法

```
uuencode [-hv] [file1 ...]</p>
```

参数：

- h 列出指令使用格式 (help)
- v 列出版本讯息

实例

将 file.uud 还原，而还原后的档名储存在 file.uud 档中。

```
uuencode file.uud
```

可以一起还原好几个档案。

```
uuencode file1.uud file2.uud
```

Linux tar命令

Linux tar命令用于备份文件。

tar是用来建立，还原备份文件的工具程序，它可以加入，解开备份文件内的文件。

语法

```
tar [-ABcdgGhiklmMoOpPrRsStuUvwWxzZ][-b <区块数目>][-C <目的目录>][-f <备份文件>][-F <Script文
```

参数：

- -A或--catenate 新增温暖件到已存在的备份文件。
- -b<区块数目>或--blocking-factor=<区块数目> 设置每笔记录的区块数目，每个区块大小为12Bytes。
- -B或--read-full-records 读取数据时重设区块大小。
- -c或--create 建立新的备份文件。
- -C<目的目录>或--directory=<目的目录> 切换到指定的目录。
- -d或--diff或--compare 对比备份文件内和文件系统上的文件的差异。
- -f<备份文件>或--file=<备份文件> 指定备份文件。
- -F<Script文件>或--info-script=<Script文件> 每次更换磁带时，就执行指定的Script文件。
- -g或--listed-incremental 处理GNU格式的大量备份。
- -G或--incremental 处理旧的GNU格式的大量备份。
- -h或--dereference 不建立符号连接，直接复制该连接所指向的原始文件。
- -i或--ignore-zeros 忽略备份文件中的0 Byte区块，也就是EOF。
- -k或--keep-old-files 解开备份文件时，不覆盖已有的文件。
- -K<文件>或--starting-file=<文件> 从指定的文件开始还原。
- -l或--one-file-system 复制的文件或目录存放的文件系统，必须与tar指令执行时所处的文件系统相同，否则不予复制。
- -L<媒体容量>或-tape-length=<媒体容量> 设置存放每体的容量，单位以1024 Bytes计算。
- -m或--modification-time 还原文件时，不变更文件的更改时间。
- -M或--multi-volume 在建立，还原备份文件或列出其中的内容时，采用多卷册模式。
- -N<日期格式>或--newer=<日期时间> 只将较指定日期更新的文件保存到备份文件里。
- -o或--old-archive或--portability 将资料写入备份文件时使用V7格式。
- -O或--stdout 把从备份文件里还原的文件输出到标准输出设备。
- -p或--same-permissions 用原来的文件权限还原文件。
- -P或--absolute-names 文件名使用绝对名称，不移除文件名称前的"/"号。

- -r或--append 新增文件到已存在的备份文件的结尾部分。
- -R或--block-number 列出每个信息在备份文件中的区块编号。
- -s或--same-order 还原文件的顺序和备份文件内的存放顺序相同。
- -S或--sparse 倘若一个文件内含大量的连续0字节，则将此文件存成稀疏文件。
- -t或--list 列出备份文件的内容。
- -T<范本文件>或--files-from=<范本文件> 指定范本文件，其内含有一个或多个范本样式，让tar解开或建立符合设置条件的文件。
- -u或--update 仅置换较备份文件内的文件更新的文件。
- -U或--unlink-first 解开压缩文件还原文件之前，先解除文件的连接。
- -v或--verbose 显示指令执行过程。
- -V<卷册名称>或--label=<卷册名称> 建立使用指定的卷册名称的备份文件。
- -w或--interactive 遭遇问题时先询问用户。
- -W或--verify 写入备份文件后，确认文件正确无误。
- -x或--extract或--get 从备份文件中还原文件。
- -X<范本文件>或--exclude-from=<范本文件> 指定范本文件，其内含有一个或多个范本样式，让ar排除符合设置条件的文件。
- -z或--gzip或--ungzip 通过gzip指令处理备份文件。
- -Z或--compress或--uncompress 通过compress指令处理备份文件。
- -<设备编号><存储密度> 设置备份用的外围设备编号及存放数据的密度。
- --after-date=<日期时间> 此参数的效果和指定"-N"参数相同。
- --atime-preserve 不变更文件的存取时间。
- --backup=<备份方式>或--backup 移除文件前先进行备份。
- --checkpoint 读取备份文件时列出目录名称。
- --concatenate 此参数的效果和指定"-A"参数相同。
- --confirmation 此参数的效果和指定"-w"参数相同。
- --delete 从备份文件中删除指定的文件。
- --exclude=<范本样式> 排除符合范本样式的问家。
- --group=<群组名称> 把加入设备文件中的文件的所属群组设成指定的群组。
- --help 在线帮助。
- --ignore-failed-read 忽略数据读取错误，不中断程序的执行。
- --new-volume-script=<Script文件> 此参数的效果和指定"-F"参数相同。
- --newer-mtime 只保存更改过的文件。
- --no-recursion 不做递归处理，也就是指定目录下的所有文件及子目录不予处理。
- --null 从null设备读取文件名称。
- --numeric-owner 以用户识别码及群组识别码取代用户名称和群组名称。
- --owner=<用户名称> 把加入备份文件中的文件的拥有者设成指定的用户。
- --posix 将数据写入备份文件时使用POSIX格式。
- --preserve 此参数的效果和指定"-ps"参数相同。
- --preserve-order 此参数的效果和指定"-A"参数相同。
- --preserve-permissions 此参数的效果和指定"-p"参数相同。

- --record-size=<区块数目> 此参数的效果和指定"-b"参数相同。
- --recursive-unlink 解开压缩文件还原目录之前，先解除整个目录下所有文件的连接。
- --remove-files 文件加入备份文件后，就将其删除。
- --rsh-command=<执行指令> 设置要在远端主机上执行的指令，以取代rsh指令。
- --same-owner 尝试以相同的文件拥有者还原问家你。
- --suffix=<备份字尾字符串> 移除文件前先行备份。
- --totals 备份文件建立后，列出文件大小。
- --use-compress-program=<执行指令> 通过指定的指令处理备份文件。
- --version 显示版本信息。
- --volno-file=<编号文件> 使用指定文件内的编号取代预设的卷册编号。

实例

压缩文件 非打包

```
# touch a.c
# tar -czvf test.tar.gz a.c   //压缩 a.c文件为test.tar.gz
a.c
```

列出压缩文件内容

```
# tar -tzvf test.tar.gz
-rw-r--r-- root/root     0 2010-05-24 16:51:59 a.c
```

解压文件

tar -xzvf test.tar.gz a.c

Linux命令大全 - 设备管理

setleds	loadkeys	rdev	dumpkeys
MAKEDEV			

Linux setleds命令

Linux setleds命令用来设定键盘上方三个 LED 的状态。在 Linux 中，每一个虚拟主控台都有独立的设定。

语法

```
setleds [-v] [-L] [-D] [-F] [{+|-}num] [{+|-}caps] [{+|-}scroll]
```

参数：

- -F：预设的选项，设定虚拟主控台的状态。
- -D：除了改变虚拟主控台的状态外，还改变预设的状态。
- -L：不改变虚拟主控台的状态，但直接改变 LED 显示的状态。这会使得 LDE 显示和目前虚拟主控台的状态不符合。我们可以在稍后用 -L 且不含其它选项的 setleds 命令回复正常状态。
- -num +num：将数字键打开或关闭。
- -caps +caps：把大小写键打开或关闭。
- -scroll +scroll：把选项键打开或关闭。

实例

将数字键打开，其馀二个灯关闭。

```
# setleds +num -caps -scroll
```

Linux loadkeys命令

Linux loadkeys命令可以根据一个键盘定义表改变 linux 键盘驱动程序转译键盘输入过程。详细的说明请参考 dumpkeys。

语法

```
loadkeys [ -d --default ] [ -h --help ] [ -q --quiet ] [ -v --verbose [ -v --verbose ]...
```

参数:

- -v --verbose：印出详细的资料，你可以重复以增加详细度。
- -q --quiet：不要显示任何讯息。
- -c --clearcompose：清除所有 composite 定义。
- -s --clearstrings：将定串定义表清除。

实例

```
定义按键组合
<pre>
# loadkeys
control alt keycode 88 = F80 //现确定键代码
string F80="w3cschool.cc" //给变变量设定值
//按下 Ctrl + D键 确定输入

//效果：按下 Ctrl +Alt + F12 输出 Lx138.Com

# dumpkeys --funcs-only //显示功能键

……省略部分结果
string F3 = "\033[[C"
string F4 = "\033[[D"
string F5 = "\033[[E"
string F6 = "\033[17~"
string F7 = "\033[18~"
string F8 = "\033[19~"
string F9 = "\033[20~"
string F10 = "\033[21~"
string F11 = "\033[23~"
string F12 = "\033[24~"
string F13 = "\033[25~"
string F14 = "\033[26~"
string F15 = "\033[28~"
string F16 = "\033[29~"
string F17 = "\033[31~"
string F18 = "\033[32~"
string F19 = "\033[33~"
string F20 = "\033[34~"
string Find = "\033[1~"
string Insert = "\033[2~"
string Remove = "\033[3~"
string Select = "\033[4~"
string Prior = "\033[5~"
string Next = "\033[6~"
string Macro = "\033[M"
string Pause = "\033[P"
string F80 = "w3cschool.cc"
```

Linux rdev命令

Linux rdev命令可以用来查询/设置内核映像文件的根设备，RAM 磁盘大小或视频模式。

不带任何参数的 rdev 命令将输出当前根文件系统的 /etc/mtab 文件行。不带任何参数的 ramsize, vidmode, 和 rootflags 将显示帮助信息。

语法

```
rdev [-rsvh ] [-o offset ] [ image [value [ offset ] ] ]</p>
```

但是随著使用者想要设定的参数的不同，底下的方式也是一样：

```
rdev [ -o offset ] [ image [ root_device [ offset ] ] ]
```

```
swapdev [ -o offset ] [ image [ swap_device [ offset ] ] ]
```

```
ramsize [ -o offset ] [ image [ size [ offset ] ] ]
```

```
videomode [ -o offset ] [ image [ mode [ offset ] ] ]
```

```
rootflags [ -o offset ] [ image [ flags [ offset ] ] ]
```

参数：

- -r： 使得 rdev 作为 ramsize 运行。
- -R： 使得 rdev 作为 rootflags 运行。
- -v： 使得 rdev 作为 vidmode 运行。
- -h： 提供帮助。

Linux dumpkeys命令

Linux dumpkeys命令用于显示键盘映射表，输出的内容可以被loadkeys命令识别,改变映射关系。

语法

```
dumpkey[选择参数]
```

参数说明:

- -i 驱动信息(键码范围、数量、状态键)
- -l 详细驱动信息
- -n 十六进制显示
- -f 显示全部信息
- -1 分行显示按键组合
- -S 设定输出格式(0：预设 1：完整 2：分行 3简单)
- --funcs-only 功能键信息
- --keys-only 键组合信息
- --compose-only 普通键信息

实例

显示功能键信息

```
# dumpkeys --funcs-only
string F1 = "\033[[A"
string F2 = "\033[[B"
string F3 = "\033[[C"
string F4 = "\033[[D"
string F5 = "\033[[E"
string F6 = "\033[17~"
string F7 = "\033[18~"
string F8 = "\033[19~"
string F9 = "\033[20~"
string F10 = "\033[21~"
string F11 = "\033[23~"
string F12 = "\033[24~"
string F13 = "\033[25~"
string F14 = "\033[26~"
string F15 = "\033[28~"
string F16 = "\033[29~"
string F17 = "\033[31~"
string F18 = "\033[32~"
string F19 = "\033[33~"
string F20 = "\033[34~"
string Find = "\033[1~"
string Insert = "\033[2~"
string Remove = "\033[3~"
string Select = "\033[4~"
string Prior = "\033[5~"
string Next = "\033[6~"
string Macro = "\033[M"
string Pause = "\033[P"
root@snail-hnlinux:~#
```

显示驱动信息

```
# dumpkeys -i
键值码范围被内核支持: 1 - 255
可绑定到键值的动作最大值: 256
实际使用的键值数: 128
其中 121 已动态分配
被内核支持的动作码值范围
0x0000 - 0x00ff
0x0100 - 0x01ff
0x0200 - 0x0213
0x0300 - 0x0313
0x0400 - 0x0405
0x0500 - 0x05ff
0x0600 - 0x0603
0x0700 - 0x0708
0x0800 - 0x08ff
0x0900 - 0x0919
0x0a00 - 0x0a08
0x0b00 - 0x0bff
0x0c00 - 0x0c08
0x0d00 - 0x0dff
0x0e00 - 0x0e0a
内核支持的功能键数：256
编写定义的最大nr: 256
实际使用的编写定义nr: 68
```

Linux MAKEDEV命令

Linux MAKEDEV命令用于新增 /dev/ 下的装置档案，多数分区已经将所有的档案都产生，故一般而言不太会需要用到这个命令。

语法

```
MAKEDEV -V
MAKEDEV [ -n ] [ -v ] update
MAKEDEV [ -n ] [ -v ] [ -d ] device ...
```

W3School Ruby教程

来源：Ruby教程

整理：飞龙

Ruby 基础

Ruby 简介

Ruby是一种纯粹的面向对象编程语言。它由日本的松本行弘（まつもとゆきひろ/Yukihiro Matsumoto）创建于1993年。

您可以在 www.ruby-lang.org 的 Ruby 邮件列表上找到松本行弘（まつもとゆきひろ/Yukihiro Matsumoto）的名字。在 Ruby 社区，松本也被称为马茨（Matz）。

Ruby 是"程序员的最佳朋友"。

Ruby 的特性与 Smalltalk、Perl 和 Python 类似。Perl、Python 和 Smalltalk 是脚本语言。Smalltalk 是一个真正的面向对象语言。Ruby，与 Smalltalk 一样，是一个完美的面向对象语言。使用 Ruby 的语法比使用 Smalltalk 的语法要容易得多。

Ruby 的特性

- Ruby 是开源的，在 Web 上免费提供，但需要一个许可证。
- Ruby 是一种通用的、解释的编程语言。
- Ruby 是一种真正的面向对象编程语言。
- Ruby 是一种类似于 Python 和 Perl 的服务器端脚本语言。
- Ruby 可以用来编写通用网关接口（CGI）脚本。
- Ruby 可以被嵌入到超文本标记语言（HTML）。
- Ruby 语法简单，这使得新的开发人员能够快速轻松地学习 Ruby。
- Ruby 与 C++ 和 Perl 等许多编程语言有着类似的语法。
- Ruby 可扩展性强，用 Ruby 编写的大程序易于维护。
- Ruby 可用于开发的 Internet 和 Intranet 应用程序。
- Ruby 可以安装在 Windows 和 POSIX 环境中。
- Ruby 支持许多 GUI 工具，比如 Tcl/Tk、GTK 和 OpenGL。
- Ruby 可以很容易地连接到 DB2、MySQL、Oracle 和 Sybase。
- Ruby 有丰富的内置函数，可以直接在 Ruby 脚本中使用。

您需要的工具

为了执行本教程中讨论的实例，您需要 RAM 至少为 2GB（推荐为 4GB）的 Intel Core i3 或 i5 的计算机。您还需要以下软件：

- Linux 或 Windows 95/98/2000/NT 或 Windows 7 操作系统
- Apache 1.3.19-5 Web 服务器
- Internet Explorer 5.0 或以上的 Web 浏览器

- Ruby 1.8.5

本教程将介绍如何使用 Ruby 创建 GUI、网络和 Web 应用程序。另外还会讨论如何扩展和嵌入 Ruby 应用程序。

接下来将学习什么？

下一章将向您介绍从哪里可以获取 Ruby 及其文档。最后，它会指示您如何安装 Ruby，并配置环境为开发 Ruby 应用程序做准备。

Ruby 环境

本地环境设置

如果您想要设置 Ruby 编程语言的环境，请阅读本章节的内容。本章将向您讲解与环境设置有关的所有重要的主题。建议先学习下面几个主题，然后再进一步深入学习其他主题：

- Linux/Unix 上的 Ruby 安装：如果您想要在 Linux/Unix 上配置开发环境，那么请查看本章节的内容。
- Windows 上的 Ruby 安装：如果您想要在 Windows 上配置开发环境，那么请查看本章节的内容。
- Ruby 命令行选项：本章节列出了所有的命令行选项，您可以和 Ruby 解释器一起使用这些命令行选项。
- Ruby 环境变量：本章节列出了所有重要的环境变量列表，设置这些环境变量以便让 Ruby 解释器工作。

流行的 Ruby 编辑器

为了编写 Ruby 程序，您需要一个编辑器：

- 如果您是在 Windows 上进行编写，那么您可以使用任何简单的文本编辑器，比如 Notepad 或 Edit plus。
- VIM（Vi IMproved）是一个简单的文本编辑器，几乎在所有的 Unix 上都是可用的，现在也能在 Windows 上使用。另外，您还可以使用您喜欢的 vi 编辑器来编写 Ruby 程序。
- RubyWin 是一个针对 Windows 的 Ruby 集成开发环境（IDE）。
- Ruby Development Environment （RDE） 对于 Windows 用户来说，也是一个很好的集成开发环境（IDE）。

交互式 Ruby（IRb）

交互式 Ruby（IRb）为体验提供了一个 shell。在 IRb shell 内，您可以逐行立即查看解释结果。

这个工具会随着 Ruby 的安装自动带有，所以您不需要做其他额外的事情，IRb 即可正常工作。

只需要在命令提示符中键入 **irb**，一个交互式 Ruby Session 将会开始，如下所示：

```
$irb
irb 0.6.1(99/09/16)
irb(main):001:0> def hello
irb(main):002:1> out = "Hello World"
irb(main):003:1> puts out
irb(main):004:1> end
nil
irb(main):005:0> hello
Hello World
nil
irb(main):006:0>
```

这里您可以先不用关心上面命令的执行内容，我们将在后续的章节中向您讲解。

接下来将学习什么？

假设现在您已经设置好 Ruby 环境，且已经做好编写第一个 Ruby 程序的准备。下一章我们将向您讲解如何编写 Ruby 程序。

Ruby 安装 - Unix

下面列出了在 Unix 机器上安装 Ruby 的步骤。

注意：在安装之前，请确保您有 root 权限。

- 下载最新版的 Ruby 压缩文件。请点击这里下载。
- 下载 Ruby 之后，解压到新创建的目录下：

```
$ tar -xvzf ruby-1.6.7.tgz
$ cd ruby-1.6.7
```

- 现在，配置并编译源代码，如下所示：

```
$ ./configure
$ make
```

- 最后，安装 Ruby 解释器，如下所示：

```
$ su -l root # 使用root用户
$ make install
$ exit       # 切换回普通用户
```

- 安装后，通过在命令行中输入以下命令来确保一切工作正常：

```
$ruby -v
ruby 1.6.7 (2002-06-04) [i386-netbsd]
```

- 如果一切工作正常，将会输出所安装的 Ruby 解释器的版本，如上所示。如果您安装了其他版本，则会显示其他不同的版本。

使用 yum 安装 Ruby

如果您的计算机已经连接到 Internet，那么最简单的安装 Ruby 的方式是使用 **yum**。在命令提示符中输入以下的命令，即可在您的计算机上安装 Ruby。

```
$ yum install ruby
```

Ruby 安装 - Windows

下面列出了在 Windows 机器上安装 Ruby 的步骤。

注意：在安装时，您可能有不同的可用版本。

- 下载最新版的 Ruby 压缩文件。请点击这里下载。
- 下载 Ruby 之后，解压到新创建的目录下：
- 双击 Ruby1.6.7.exe 文件，启动 Ruby 安装向导。
- 点击 Next，继续向导的 Important Information 页面，直到 Ruby 安装程序完成 Ruby 安装为止。

如果您的安装没有适当地配置环境变量，接下来您可能需要进行环境变量的配置。

- 如果您使用的是 Windows 9x，那么请在您的 c:\autoexec.bat 中添加：set PATH="D:(ruby 安装目录)\bin;%PATH%"
- Windows NT/2000 用户需要修改注册表。
    - 点击控制面板|系统性能|环境变量。
    - 在系统变量下，选择 Path，并点击 EDIT。
    - 在变量值列表的末尾添加 Ruby 目录，并点击 OK。
    - 在系统变量下，选择 PATHEXT，并点击 EDIT。
    - 添加 .RB 和 .RBW 到变量值列表中，并点击 OK。
- 安装后，通过在命令行中输入以下命令来确保一切工作正常：

```
$ruby -v
ruby 1.6.7
```

- 如果一切工作正常，将会输出所安装的 Ruby 解释器的版本，如上所示。如果您安装了其他版本，则会显示其他不同的版本。

Ruby 命令行选项

Ruby 一般是从命令行运行，方式如下：

```
$ ruby [ options ] [.] [ programfile ] [ arguments ... ]
```

解释器可以通过下列选项被调用，来控制解释器的环境和行为。

选项	描述
-a	与 -n 或 -p 一起使用时，可以打开自动拆分模式(auto split mode)。请查看 -n 和 -p 选项。
-c	只检查语法，不执行程序。
-C dir	在执行前改变目录（等价于 -X）。
-d	启用调试模式（等价于 -debug）。
-F pat	指定 pat 作为默认的分离模式（$;）。
-e prog	指定 prog 作为程序在命令行中执行。可以指定多个 -e 选项，用来执行多个程序。
-h	显示命令行选项的一个概览。
-i [ext]	把文件内容重写为程序输出。原始文件会被加上扩展名 ext 保存下来。如果未指定 ext，原始文件会被删除。
-I dir	添加 dir 作为加载库的目录。
-K [kcode]	指定多字节字符集编码。e 或 E 对应 EUC（extended Unix code），s 或 S 对应 SJIS（Shift-JIS），u 或 U 对应 UTF-8，a、A、n 或 N 对应 ASCII。
-l	启用自动行尾处理。从输入行取消一个换行符，并向输出行追加一个换行符。
-n	把代码放置在一个输入循环中（就像在 while gets; ... end 中一样）。
-0[octal]	设置默认的记录分隔符（$/）为八进制。如果未指定 octal 则默认为 \0。
-p	把代码放置在一个输入循环中。在每次迭代后输出变量 $_ 的值。
-r lib	使用 *require* 来加载 *lib* 作为执行前的库。
-s	解读程序名称和文件名参数之间的匹配模式 -xxx 的任何参数作为开关，并定义相应的变量。
-T [level]	设置安全级别，执行不纯度测试（如果未指定 level，则默认值为 1）。
-v	显示版本，并启用冗余模式。
-w	启用冗余模式。如果未指定程序文件，则从 STDIN 读取。

-x [dir]	删除 #!ruby 行之前的文本。如果指定了 *dir*，则把目录改变为 *dir*。
-X dir	在执行前改变目录（等价于 -C）。
-y	启用解析器调试模式。
--copyright	显示版权声明。
--debug	启用调试模式（等价于 -d）。
--help	显示命令行选项的一个概览（等价于 -h）。
--version	显示版本。
--verbose	启用冗余模式（等价于 -v）。设置 $VERBOSE 为 true。
--yydebug	启用解析器调试模式（等价于 -y）。

单字符的命令行选项可以组合使用。下面两行表达了同样的意思：

```
$ruby -ne 'print if /Ruby/' /usr/share/bin

$ruby -n -e 'print if /Ruby/' /usr/share/bin
```

Ruby 环境变量

Ruby 解释器使用下列环境变量来控制它的行为。ENV 对象包含了所有当前设置的环境变量列表。

变量	描述
DLN_LIBRARY_PATH	动态加载模块搜索的路径。
HOME	当没有参数传递给 Dir::chdir 时，要移动到的目录。也用于 File::expand_path 来扩展 "~"。
LOGDIR	当没有参数传递给 Dir::chdir 且未设置环境变量 HOME 时，要移动到的目录。
PATH	执行子进程的搜索路径，以及在指定 -S 选项后，Ruby 程序的搜索路径。每个路径用冒号分隔（在 DOS 和 Windows 中用分号分隔）。
RUBYLIB	库的搜索路径。每个路径用冒号分隔（在 DOS 和 Windows 中用分号分隔）。
RUBYLIB_PREFIX	用于修改 RUBYLIB 搜索路径，通过使用格式 path1;path2 或 path1path2，把库的前缀 path1 替换为 path2。
RUBYOPT	传给 Ruby 解释器的命令行选项。在 taint 模式时被忽略（其中，$SAFE 大于 0）。
RUBYPATH	指定 -S 选项后，Ruby 程序的搜索路径。优先级高于 PATH。在 taint 模式时被忽略（其中，$SAFE 大于 0）。
RUBYSHELL	指定执行命令时所使用的 shell。如果未设置该环境变量，则使用 SHELL 或 COMSPEC。

对于 Unix，使用 **env** 命令来查看所有环境变量的列表。

```
HOSTNAME=ip-72-167-112-17.ip.secureserver.net
RUBYPATH=/usr/bin
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
SSH_CLIENT=122.169.131.179 1742 22
SSH_TTY=/dev/pts/1
USER=amrood
JRE_HOME=/usr/java/jdk/jre
J2RE_HOME=/usr/java/jdk/jre
PATH=/usr/local/bin:/bin:/usr/bin:/home/guest/bin
MAIL=/var/spool/mail/guest
PWD=/home/amrood
INPUTRC=/etc/inputrc
JAVA_HOME=/usr/java/jdk
LANG=C
HOME=/root
SHLVL=2
JDK_HOME=/usr/java/jdk
LOGDIR=/usr/log/ruby
LOGNAME=amrood
SSH_CONNECTION=122.169.131.179 1742 72.167.112.17 22
LESSOPEN=|/usr/bin/lesspipe.sh %s
RUBYLIB=/usr/lib/ruby
G_BROKEN_FILENAMES=1
_=/bin/env
```

Ruby 语法

让我们编写一个简单的 Ruby 程序。所有的 Ruby 文件扩展名都是 **.rb**。所以，把下面的源代码放在 test.rb 文件中。

```
#!/usr/bin/ruby -w

puts "Hello, Ruby!";
```

在这里，假设您的 /usr/bin 目录下已经有可用的 Ruby 解释器。现在，尝试运行这个程序，如下所示：

```
$ ruby test.rb
```

这将会产生下面的结果：

```
Hello, Ruby!
```

您已经看到了一个简单的 Ruby 程序，现在让我们看看一些 Ruby 语法相关的基本概念：

Ruby 程序中的空白

在 Ruby 代码中的空白字符，如空格和制表符一般会被忽略，除非当它们出现在字符串中时才不会被忽略。然而，有时候它们用于解释模棱两可的语句。当启用 -w 选项时，这种解释会产生警告。

实例：

```
a + b 被解释为 a+b （这是一个局部变量）
a  +b 被解释为 a(+b) （这是一个方法调用）
```

Ruby 程序中的行尾

Ruby 把分号和换行符解释为语句的结尾。但是，如果 Ruby 在行尾遇到运算符，比如 +、- 或反斜杠，它们表示一个语句的延续。

Ruby 标识符

标识符是变量、常量和方法的名称。Ruby 标识符是大小写敏感的。这意味着 Ram 和 RAM 在 Ruby 中是两个不同的标识符。

Ruby 标识符的名称可以包含字母、数字和下划线字符（ _ ）。

保留字

下表列出了 Ruby 中的保留字。这些保留字不能作为常量或变量的名称。但是，它们可以作为方法名。

BEGIN	do	next	then
END	else	nil	true
alias	elsif	not	undef
and	end	or	unless
begin	ensure	redo	until
break	false	rescue	when
case	for	retry	while
class	if	return	while
def	in	self	__FILE__
defined?	module	super	__LINE__

Ruby 中的 Here Document

"Here Document" 是指建立多行字符串。在 << 之后，您可以指定一个字符串或标识符来终止字符串，且当前行之后直到终止符为止的所有行是字符串的值。

如果终止符用引号括起，引号的类型决定了面向行的字符串类型。请注意<< 和终止符之间必须没有空格。

下面是不同的实例：

```
#!/usr/bin/ruby -w

print <<EOF
    This is the first way of creating
    here document ie. multiple line string.
EOF

print <<"EOF";                # 与上面相同
    This is the second way of creating
    here document ie. multiple line string.
EOF

print <<`EOC`                 # 执行命令
    echo hi there
    echo lo there
EOC

print <<"foo", <<"bar"          # 您可以把它们进行堆叠
    I said foo.
foo
    I said bar.
bar
```

这将产生以下结果：

```
    This is the first way of creating
    her document ie. multiple line string.
    This is the second way of creating
    her document ie. multiple line string.
hi there
lo there
        I said foo.
        I said bar.
```

Ruby *BEGIN* 语句

语法

```
BEGIN {
   code
}
```

声明 *code* 会在程序运行之前被调用。

实例

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

BEGIN {
   puts "Initializing Ruby Program"
}
```

这将产生以下结果：

```
Initializing Ruby Program
This is main Ruby Program
```

Ruby *END* 语句

语法

```
END {
   code
}
```

声明 *code* 会在程序的结尾被调用。

实例

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

END {
   puts "Terminating Ruby Program"
}
BEGIN {
   puts "Initializing Ruby Program"
}
```

这将产生以下结果：

```
Initializing Ruby Program
This is main Ruby Program
Terminating Ruby Program
```

Ruby 注释

注释会对 Ruby 解释器隐藏一行，或者一行的一部分，或者若干行。您可以在行首使用字符（ # ）：

```
# 我是注释，请忽略我。
```

或者，注释可以跟着语句或表达式的同一行的后面：

```
name = "Madisetti" # 这也是注释
```

您可以注释多行，如下所示：

```
# 这是注释。
# 这也是注释。
# 这也是注释。
# 这还是注释。
```

下面是另一种形式。这种块注释会对解释器隐藏 =begin/=end 之间的行：

```
=begin
这是注释。
这也是注释。
这也是注释。
这还是注释。
=end
```

Ruby 数据类型

本章节我们将为大家介绍 Ruby 的基本数据类型。

Ruby支持的数据类型包括基本的Number、String、Ranges、Symbols，以及true、false和nil这几个特殊值，同时还有两种重要的数据结构——Array和Hash。

数值类型(Number)

1、整型(Integer)

整型分两种，如果在31位以内（四字节），那为Fixnum实例。如果超过，即为Bignum实例。

整数范围从 -2³⁰ 到 2³⁰⁻¹ 或 -2⁶² 到 2⁶²⁻¹。在这个范围内的整数是类 *Fixnum* 的对象，在这个范围外的整数存储在类 *Bignum* 的对象中。

您可以在整数前使用一个可选的前导符号，一个可选的基础指标（0 对应 octal，0x 对应 hex，0b 对应 binary），后跟一串数字。下划线字符在数字字符串中被忽略。

您可以获取一个 ASCII 字符或一个用问号标记的转义序列的整数值。

实例

```
123                  # Fixnum 十进制
1_234                # Fixnum 带有下划线的十进制
-500                 # 负的 Fixnum
0377                 # 八进制
0xff                 # 十六进制
0b1011               # 二进制
?a                   # 'a' 的字符编码
?\n                  # 换行符（0x0a）的编码
12345678901234567890 # Bignum
```

```
#整型 Integer 以下是一些整型字面量
#字面量（literal）：代码中能见到的值，数值，bool值，字符串等都叫字面量
#如以下的0,1_000_000,0xa等
a1=0

#带千分符的整型
a2=1_000_000

#其它进制的表示
a3=0xa
puts a1,a2
puts a3

#puts print 都是向控制台打印字符，其中puts带回车换行符
=begin
这是注释，称作：嵌入式文档注释
类似C#中的/**/
=end
```

浮点型

Ruby 支持浮点数。它们是带有小数的数字。浮点数是类 *Float* 的对象，且可以是下列中任意一个。

实例

```
123.4                # 浮点值
1.0e6                # 科学记数法
4E20                 # 不是必需的
4e+20                # 指数前的符号
```

```
#浮点型
f1=0.0
f2=2.1
f3=1000000.1
puts f3
```

算术操作

加减乘除操作符：+-/；指数操作符为*

指数不必是整数，例如

```
#指数算术
puts 2**(1/4)#1与4的商为0，然后2的0次方为1
puts 16**(1/4.0)#1与4.0的商为0.25（四分之一），然后开四次方根
```

字符串类型

Ruby 字符串简单地说是一个 8 位字节序列，它们是类 String 的对象。

双引号标记的字符串允许替换和使用反斜线符号，单引号标记的字符串不允许替换，且只允许使用 \ 和 \' 两个反斜线符号。

实例

```
#!/usr/bin/ruby -w

puts 'escape using "\\"';
puts 'That\'s right';
```

这将产生以下结果：

```
escape using "\"
That's right
```

您可以使用序列 **#{ expr }** 替换任意 Ruby 表达式的值为一个字符串。在这里，expr 可以是任意的 Ruby 表达式。

```
#!/usr/bin/ruby -w

puts "Multiplication Value : #{24*60*60}";
```

这将产生以下结果：

```
Multiplication Value : 86400
```

```
#!/usr/bin/ruby -w

name="Ruby"
puts name
puts "#{name+",ok"}"
```

输出结果为：

```
Ruby
Ruby,ok
```

反斜线符号

下表列出了 Ruby 支持的反斜线符号：

符号	表示的字符
\n	换行符 (0x0a)
\r	回车符 (0x0d)
\f	换页符 (0x0c)
\b	退格键 (0x08)
\a	报警符 Bell (0x07)
\e	转义符 (0x1b)
\s	空格符 (0x20)
\nnn	八进制表示法 (n 是 0-7)
\xnn	十六进制表示法 (n 是 0-9、a-f 或 A-F)
\cx, \C-x	Control-x
\M-x	Meta-x (c | 0x80)
\M-\C-x	Meta-Control-x
\x	字符 x

如需了解更多有关 Ruby 字符串的细节，请查看 Ruby 字符串（String）。

数组

数组字面量通过[]中以逗号分隔定义，且支持range定义。

- （1）数组通过[]索引访问
- （2）通过赋值操作插入、删除、替换元素
- （3）通过+，－号进行合并和删除元素，且集合做为新集合出现
- （4）通过<<号向原数据追加元素
- （5）通过*号重复数组元素
- （6）通过｜和&符号做并集和交集操作（注意顺序）

实例：

```
#!/usr/bin/ruby

ary = [  "fred", 10, 3.14, "This is a string", "last element", ]
ary.each do |i|
   puts i
end
```

这将产生以下结果：

```
fred
10
3.14
This is a string
last element
```

如需了解更多有关 Ruby 数组的细节，请查看 Ruby 数组（Array）。

哈希类型

Ruby 哈希是在大括号内放置一系列键/值对，键和值之间使用逗号和序列 => 分隔。尾部的逗号会被忽略。

实例

```
#!/usr/bin/ruby

hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" => 0x00f }
hsh.each do |key, value|
   print key, " is ", value, "\n"
end
```

这将产生以下结果：

```
green is 240
red is 3840
blue is 15
```

如需了解更多有关 Ruby 哈希的细节，请查看 Ruby 哈希（Hash）。

范围类型

一个范围表示一个区间。

范围是通过设置一个开始值和一个结束值来表示。范围可使用 s..e 和 s...e 来构造，或者通过 Range.new 来构造。

使用 .. 构造的范围从开始值运行到结束值（包含结束值）。使用 ... 构造的范围从开始值运行到结束值（不包含结束值）。当作为一个迭代器使用时，范围会返回序列中的每个值。

范围 (1..5) 意味着它包含值 1, 2, 3, 4, 5，范围 (1...5) 意味着它包含值 1, 2, 3, 4 。

实例

```
#!/usr/bin/ruby

(10..15).each do |n|
   print n, ' '
end
```

这将产生以下结果：

```
10 11 12 13 14 15
```

如需了解更多有关 Ruby 范围的细节，请查看 Ruby 范围（Range）。

Ruby 类和对象

Ruby 是一种完美的面向对象编程语言。面向对象编程语言的特性包括：

- 数据封装
- 数据抽象
- 多态性
- 继承

这些特性将在 面向对象的 Ruby 中进行讨论。

一个面向对象的程序，涉及到的类和对象。类是个别对象创建的蓝图。在面向对象的术语中，您的自行车是自行车类的一个实例。

以车辆为例，它包括车轮（wheels）、马力（horsepower）、燃油或燃气罐容量（fuel or gas tank capacity）。这些属性形成了车辆（Vehicle）类的数据成员。借助这些属性您能把一个车辆从其他车辆中区分出来。

车辆也能包含特定的函数，比如暂停（halting）、驾驶（driving）、超速（speeding）。这些函数形成了车辆（Vehicle）类的数据成员。因此，您可以定义类为属性和函数的组合。

类 Vehicle 的定义如下：

```
Class Vehicle
{
   Number no_of_wheels
   Number horsepower
   Characters type_of_tank
   Number Capacity
   Function speeding
   {
   }
   Function driving
   {
   }
   Function halting
   {
   }
}
```

通过给这些数据成员分配不同的值，您可以创建类 Vehicle 的不同实例。例如，一架飞机有三个轮子，马力 1,000，燃油罐容量为 100 升。以同样的方式，一辆汽车有四个轮子，马力 200，煤气罐容量为 25 升。

在 Ruby 中定义类

为了使用 Ruby 实现面向对象编程，您需要先学习如何在 Ruby 中创建对象和类。

在 Ruby 中，类总是以关键字 *class* 开始，后跟类的名称。类名的首字母应该大写。类 *Customer* 如下所示：

```
class Customer
end
```

您可以使用关键字 *end* 终止一个类。类 中的所有数据成员都是介于类定义和 *end* 关键字之间。

Ruby 类中的变量

Ruby 提供了四种类型的变量：

- 局部变量：局部变量是在方法中定义的变量。局部变量在方法外是不可用的。在后续的章节中，您将看到有关方法的更多细节。局部变量以小写字母或 _ 开始。
- 实例变量：实例变量可以跨任何特定的实例或对象中的方法使用。这意味着，实例变量可以从对象到对象的改变。实例变量在变量名之前放置符号（@）。
- 类变量：类变量可以跨不同的对象使用。类变量属于类，且是类的一个属性。类变量在变量名之前放置符号（@@）。
- 全局变量：类变量不能跨类使用。如果您想要有一个可以跨类使用的变量，您需要定义全局变量。全局变量总是以美元符号（$）开始。

实例

使用类变量 @@no_of_customers，您可以判断被创建的对象数量，这样可以确定客户数量。

```
class Customer
   @@no_of_customers=0
end
```

在 Ruby 中使用 *new* 方法创建对象

对象是类的实例。现在您将学习如何在 Ruby 中创建类的对象。在 Ruby 中，您可以使用类的方法 *new* 创建对象。

方法 *new* 是一种独特的方法，在 Ruby 库中预定义。new 方法属于类方法。

下面的实例创建了类 Customer 的两个对象 cust1 和 cust2：

```
cust1 = Customer. new
cust2 = Customer. new
```

在这里，cust1 和 cust2 是两个对象的名称。对象名称后跟着等号（=），等号后跟着类名，然后是点运算符和关键字 *new*。

自定义方法来创建 Ruby 对象

您可以给方法 *new* 传递参数，这些参数可用于初始化类变量。

当您想要声明带参数的 *new* 方法时，您需要在创建类的同时声明方法 *initialize*。

initialize 方法是一种特殊类型的方法，将在调用带参数的类的 *new* 方法时执行。

下面的实例创建了 initialize 方法：

```
class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
end
```

在本实例中，您可以声明带有 **id**、**name**、**addr** 作为局部变量的 *initialize*方法。在这里，*def* 和 *end* 用于定义 Ruby 方法 *initialize*。在后续的章节中，您将学习有关方法的更多细节。

在 *initialize* 方法中，把这些局部变量的值传给实例变量 @cust_id、@cust_name 和 @cust_addr。在这里，局部变量的值是随着 new 方法进行传递的。

现在，您可以创建对象，如下所示：

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

Ruby 类中的成员函数

在 Ruby 中，函数被称为方法。类中的每个方法是以关键字 *def* 开始，后跟方法名。

方法名总是以小写字母开头。在 Ruby 中，您可以使用关键字 *end* 来结束一个方法。

下面的实例定义了一个 Ruby 方法：

```
class Sample
   def function
      statement 1
      statement 2
   end
end
```

在这里，*statement 1* 和 *statement 2* 是类 Sample 内的方法 *function* 的主体的组成部分。这些语句可以是任何有效的 Ruby 语句。例如，我们可以使用方法 *puts* 来输出 *Hello Ruby*，如下所示：

```
class Sample
   def hello
      puts "Hello Ruby!"
   end
end
```

下面的实例将创建类 Sample 的一个对象，并调用 *hello* 方法：

```
#!/usr/bin/ruby

class Sample
   def hello
      puts "Hello Ruby!"
   end
end

# 使用上面的类来创建对象
object = Sample. new
object.hello
```

这将会产生下面的结果：

```
Hello Ruby!
```

简单的案例研究

如果您想要做更多有关类和对象的练习，这里有一个案例研究：

Ruby 类案例

Ruby 类案例

下面将创建一个名为 Customer 的 Ruby 类，您将声明两个方法：

- *display_details*：该方法用于显示客户的详细信息。
- *total_no_of_customers*：该方法用于显示在系统中创建的客户总数量。

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
    def total_no_of_customers()
       @@no_of_customers += 1
       puts "Total number of customers: #@@no_of_customers"
    end
end
```

display_details 方法包含了三个 puts 语句，显示了客户 ID、客户名字和客户地址。其中，puts 语句：

```
puts "Customer id #@cust_id"
```

将在一个单行上显示文本 Customer id，后跟变量 @cust_id 的值。

当您想要在一个单行上显示实例变量的文本和值时，您需要在 puts 语句的变量名前面放置符号（#）。文本和带有符号（#）的实例变量应使用双引号标记。

第二个方法，total*no_of_customers*，包含了类变量 @@*no_of_customers*。表达式 @@*no_of* customers+=1 在每次调用方法 total_no_of_customers 时，把变量 no_of_customers 加 1。通过这种方式，您将得到类变量中的客户总数量。

现在创建两个客户，如下所示：

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

在这里，我们创建了 Customer 类的两个对象，cust1 和 cust2，并向 new 方法传递必要的参数。当 initialize 方法被调用时，对象的必要属性被初始化。

一旦对象被创建，您需要使用两个对象来调用类的方法。如果您想要调用方法或任何数据成员，您可以编写代码，如下所示：

```
cust1.display_details()
cust1.total_no_of_customers()
```

对象名称后总是跟着一个点号，接着是方法名称或数据成员。我们已经看到如何使用 cust1 对象调用两个方法。使用 cust2 对象，您也可以调用两个方法，如下所示：

```
cust2.display_details()
cust2.total_no_of_customers()
```

保存并执行代码

现在，把所有的源代码放在 main.rb 文件中，如下所示：

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
   end
   def total_no_of_customers()
      @@no_of_customers += 1
      puts "Total number of customers: #@@no_of_customers"
   end
end

# 创建对象
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# 调用方法
cust1.display_details()
cust1.total_no_of_customers()
cust2.display_details()
cust2.total_no_of_customers()
```

接着，运行程序，如下所示：

```
$ ruby main.rb
```

这将产生以下结果：

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Total number of customers: 1
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
Total number of customers: 2
```

Ruby 变量

变量是持有可被任何程序使用的任何数据的存储位置。

Ruby 支持五种类型的变量。您已经在前面的章节中大概了解了这些变量，本章节将为您详细讲解这五种类型的变量。

Ruby 全局变量

全局变量以 $ 开头。未初始化的全局变量的值为 *nil*，在使用 -w 选项后，会产生警告。

给全局变量赋值会改变全局状态，所以不建议使用全局变量。

下面的实例显示了全局变量的用法。

```
#!/usr/bin/ruby

$global_variable = 10
class Class1
  def print_global
     puts "Global variable in Class1 is #$global_variable"
  end
end
class Class2
  def print_global
     puts "Global variable in Class2 is #$global_variable"
  end
end

class1obj = Class1.new
class1obj.print_global
class2obj = Class2.new
class2obj.print_global
```

在这里，$global_variable 是全局变量。这将产生以下结果：

注意：在 Ruby 中，您可以通过在变量或常量前面放置 # 字符，来访问任何变量或常量的值。

```
Global variable in Class1 is 10
Global variable in Class2 is 10
```

Ruby 实例变量

实例变量以 @ 开头。未初始化的实例变量的值为 *nil*，在使用 -w 选项后，会产生警告。

下面的实例显示了实例变量的用法。

```
#!/usr/bin/ruby

class Customer
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
end

# 创建对象
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# 调用方法
cust1.display_details()
cust2.display_details()
```

在这里，@cust_id、@cust_name 和 @cust_addr 是实例变量。这将产生以下结果：

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
```

Ruby 类变量

类变量以 @@ 开头，且必须初始化后才能在方法定义中使用。

引用一个未初始化的类变量会产生错误。类变量在定义它的类或模块的子类或子模块中可共享使用。

在使用 -w 选项后，重载类变量会产生警告。

下面的实例显示了类变量的用法。

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
    def total_no_of_customers()
       @@no_of_customers += 1
       puts "Total number of customers: #@@no_of_customers"
    end
end

# 创建对象
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# 调用方法
cust1.total_no_of_customers()
cust2.total_no_of_customers()
```

在这里，@@no_of_customers 是类变量。这将产生以下结果：

```
Total number of customers: 1
Total number of customers: 2
```

Ruby 局部变量

局部变量以小写字母或下划线 _ 开头。局部变量的作用域从 class、module、def 或 do 到相对应的结尾或者从左大括号到右大括号 {}。

当调用一个未初始化的局部变量时，它被解释为调用一个不带参数的方法。

对未初始化的局部变量赋值也可以当作是变量声明。变量会一直存在，直到当前域结束为止。局部变量的生命周期在 Ruby 解析程序时确定。

在上面的实例中，局部变量是 id、name 和 addr。

Ruby 常量

常量以大写字母开头。定义在类或模块内的常量可以从类或模块的内部访问，定义在类或模块外的常量可以被全局访问。

常量不能定义在方法内。引用一个未初始化的常量会产生错误。对已经初始化的常量赋值会产生警告。

```
#!/usr/bin/ruby

class Example
   VAR1 = 100
   VAR2 = 200
   def show
       puts "Value of first Constant is #{VAR1}"
       puts "Value of second Constant is #{VAR2}"
   end
end

# 创建对象
object=Example.new()
object.show
```

在这里，VAR1 和 VAR2 是常量。这将产生以下结果：

```
Value of first Constant is 100
Value of second Constant is 200
```

Ruby 伪变量

它们是特殊的变量，有着局部变量的外观，但行为却像常量。您不能给这些变量赋任何值。

- **self:** 当前方法的接收器对象。
- **true:** 代表 true 的值。
- **false:** 代表 false 的值。
- **nil:** 代表 undefined 的值。
- **FILE:** 当前源文件的名称。
- **LINE:** 当前行在源文件中的编号。

Ruby 运算符

Ruby 支持一套丰富的运算符。大多数运算符实际上是方法调用。例如，a + b 被解释为 a.+(b)，其中指向变量 a 的 + 方法被调用，b 作为方法调用的参数。

对于每个运算符（+ - / % * & | ^ << >> && ||），都有一个相对应的缩写赋值运算符（+= -= 等等）。

Ruby 算术运算符

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
+	加法 - 把运算符两边的操作数相加	a + b 将得到 30
-	减法 - 把左操作数减去右操作数	a - b 将得到 -10
*	乘法 - 把运算符两边的操作数相乘	a * b 将得到 200
/	除法 - 把左操作数除以右操作数	b / a 将得到 2
%	求模 - 把左操作数除以右操作数，返回余数	b % a 将得到 0
**	指数 - 执行指数计算	a**b 将得到 10 的 20 次方

Ruby 比较运算符

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(a == b) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(a != b) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(a > b) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(a < b) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(a >= b) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(a <= b) 为真。
<=>	联合比较运算符。如果第一个操作数等于第二个操作数则返回 0，如果第一个操作数大于第二个操作数则返回 1，如果第一个操作数小于第二个操作数则返回 -1。	(a <=> b) 返回 -1。
===	用于测试 *case* 语句的 when 子句内的相等。	(1...10) === 5 返回 true。
.eql?	如果接收器和参数具有相同的类型和相等的值，则返回 true。	1 == 1.0 返回 true，但是 1**.eql?**(1.0) 返回 false。
equal?	如果接收器和参数具有相同的对象 id，则返回 true。	如果 aObj 是 bObj 的副本，那么 aObj == bObj 返回 true，a**.equal?**bObj 返回 false，但是 a**.equal?**aObj 返回 true。

Ruby 赋值运算符

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
=	简单的赋值运算符，把右操作数的值赋给左操作数	c = a + b 将把 a + b 的值赋给 c
+=	加且赋值运算符，把右操作数加上左操作数的结果赋值给左操作数	c += a 相当于 c = c + a
-=	减且赋值运算符，把左操作数减去右操作数的结果赋值给左操作数	c -= a 相当于 c = c - a
*=	乘且赋值运算符，把右操作数乘以左操作数的结果赋值给左操作数	c = *a* 相当于 *c* = *c* a
/=	除且赋值运算符，把左操作数除以右操作数的结果赋值给左操作数	c /= a 相当于 c = c / a
%=	求模且赋值运算符，求两个操作数的模赋值给左操作数	c %= a 相当于 c = c % a
**=	指数且赋值运算符，执行指数计算，并赋值给左操作数	c **= a 相当于 c = c ** a

Ruby 并行赋值

Ruby 也支持变量的并行赋值。这使得多个变量可以通过一行的 Ruby 代码进行初始化。例如：

```
a = 10
b = 20
c = 30
```

使用并行赋值可以更快地声明：

```
a, b, c = 10, 20, 30
```

并行赋值在交换两个变量的值时也很有用：

```
a, b = b, c
```

Ruby 位运算符

位运算符作用于位，并逐位执行操作。

假设如果 a = 60，且 b = 13，现在以二进制格式，它们如下所示：

```
a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011
```

下表列出了 Ruby 支持的位运算符。

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(a & b) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(a | b) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(a ^ b) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果。	(~a) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	a << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	a >> 2 将得到 15，即为 0000 1111

Ruby 逻辑运算符

下表列出了 Ruby 支持的逻辑运算符。

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
and	称为逻辑与运算符。如果两个操作数都为真，则条件为真。	(a and b) 为真。
or	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(a or b) 为真。
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(a && b) 为真。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(a || b) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(a && b) 为假。
not	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	not(a && b) 为假。

Ruby 三元运算符

有一个以上的操作称为三元运算符。第一个计算表达式的真假值，然后根据这个结果决定执行后边两个语句中的一个。条件运算符的语法如下：

运算符	描述	实例
? :	条件表达式	如果条件为真 ? 则值为 X : 否则值为 Y

Ruby 范围运算符

在 Ruby 中，序列范围用于创建一系列连续的值 - 包含起始值、结束值（视情况而定）和它们之间的值。

在 Ruby 中，这些序列是使用 ".." 和 "..." 范围运算符来创建的。两点形式创建的范围包含起始值和结束值，三点形式创建的范围只包含起始值不包含结束值。

运算符	描述	实例
..	创建一个从开始点到结束点的范围（包含结束点）	1..10 创建从 1 到 10 的范围
...	创建一个从开始点到结束点的范围（不包含结束点）	1...10 创建从 1 到 9 的范围

Ruby defined? 运算符

defined? 是一个特殊的运算符，以方法调用的形式来判断传递的表达式是否已定义。它返回表达式的描述字符串，如果表达式未定义则返回 *nil*。

下面是 defined? 运算符的各种用法：

用法 1

```
defined? variable # 如果 variable 已经初始化，则为 True
```

例如：

```
foo = 42
defined? foo    # => "local-variable"
defined? $_     # => "global-variable"
defined? bar    # => nil（未定义）
```

用法 2

```
defined? method_call # 如果方法已经定义，则为 True
```

例如：

```
defined? puts        # => "method"
defined? puts(bar)   # => nil（在这里 bar 未定义）
defined? unpack      # => nil（在这里未定义）
```

用法 3

```
# 如果存在可被 super 用户调用的方法，则为 True
defined? super
```

例如：

```
defined? super     # => "super"（如果可被调用）
defined? super     # => nil（如果不可被调用）
```

用法 4

```
defined? yield   # 如果已传递代码块，则为 True
```

例如：

```
defined? yield    # => "yield"（如果已传递块）
defined? yield    # => nil（如果未传递块）
```

Ruby 点运算符 "." 和双冒号运算符 "::"

您可以通过在方法名称前加上模块名称和一条下划线来调用模块方法。您可以使用模块名称和两个冒号来引用一个常量。

:: 是一元运算符，允许在类或模块内定义常量、实例方法和类方法，可以从类或模块外的任何地方进行访问。

请记住：在 Ruby 中，类和方法也可以被当作常量。

您只需要在表达式的常量名前加上 :: 前缀，即可返回适当的类或模块对象。

如果未使用前缀表达式，则默认使用主 Object 类。

下面是两个实例：

```
MR_COUNT = 0        # 定义在主 Object 类上的常量
module Foo
  MR_COUNT = 0
  ::MR_COUNT = 1    # 设置全局计数为 1
  MR_COUNT = 2      # 设置局部计数为 2
end
puts MR_COUNT       # 这是全局常量
puts Foo::MR_COUNT  # 这是 "Foo" 的局部常量
```

第二个实例：

```
CONST = ' out there'
class Inside_one
   CONST = proc {' in there'}
   def where_is_my_CONST
      ::CONST + ' inside one'
   end
end
class Inside_two
   CONST = ' inside two'
   def where_is_my_CONST
      CONST
   end
end
puts Inside_one.new.where_is_my_CONST
puts Inside_two.new.where_is_my_CONST
puts Object::CONST + Inside_two::CONST
puts Inside_two::CONST + CONST
puts Inside_one::CONST
puts Inside_one::CONST.call + Inside_two::CONST
```

Ruby 运算符的优先级

下表按照运算符的优先级从高到低列出了所有的运算符。

方法	运算符	描述
是	::	常量解析运算符
是	[ ] [ ]=	元素引用、元素集合
是	**	指数
是	! ~ + -	非、补、一元加、一元减（最后两个的方法名为 +@ 和 -@）
是	* / %	乘法、除法、求模
是	+ -	加法和减法
是	>> <<	位右移、位左移
是	&	位与
是	^ |	位异或、位或
是	<= < > >=	比较运算符
是	<=> == === != =~ !~	相等和模式匹配运算符（!= 和 !~ 不能被定义为方法）
	&&	逻辑与
	||	逻辑或
	.. ...	范围（包含、不包含）
	? :	三元 if-then-else
	= %= { /= -= += |= &= >>= <<= *= &&= ||= **=	赋值
	defined?	检查指定符号是否已定义
	not	逻辑否定
	or and	逻辑组成

注意：在方法列标识为 是 的运算符实际上是方法，因此可以被重载。

Ruby 注释

注释是在运行时会被忽略的 Ruby 代码内的注释行。单行注释以 # 字符开始，直到该行结束，如下所示：

```
#!/usr/bin/ruby -w

# 这是一个单行注释。

puts "Hello, Ruby!"
```

当执行时，上面的程序会产生以下结果：

```
Hello, Ruby!
```

Ruby 多行注释

您可以使用 **=begin** 和 **=end** 语法注释多行，如下所示：

```
#!/usr/bin/ruby -w

puts "Hello, Ruby!"

=begin
这是一个多行注释。
可扩展至任意数量的行。
但 =begin 和 =end 只能出现在第一行和最后一行。
=end
```

当执行时，上面的程序会产生以下结果：

```
Hello, Ruby!
```

请确保尾部的注释离代码有足够的距离，以便容易区分注释和代码。如果块中超过一条尾部注释，请对齐它们。例如：

```
@counter      # 跟踪页面被击中的次数
@siteCounter  # 跟踪所有页面被击中的次数
```

Ruby 判断

Ruby 提供了其他现代语言中很常见的条件结构。在这里，我们将解释所有的条件语句和 Ruby 中可用的修饰符。

Ruby *if...else* 语句

语法

```
if conditional [then]
      code...
[elsif conditional [then]
      code...]...
[else
      code...]
end
```

if 表达式用于条件执行。值 *false* 和 *nil* 为假，其他值都为真。请注意，Ruby 使用 elsif，不是使用 else if 和 elif。

如果 *conditional* 为真，则执行 *code*。如果 *conditional* 不为真，则执行 else 子句中指定的 *code*。

if 表达式的 *conditional* 通过保留字 *then*、一个换行符或一个分号，来与代码分离开。

实例

```
#!/usr/bin/ruby

x=1
if x > 2
   puts "x is greater than 2"
elsif x <= 2 and x!=0
   puts "x is 1"
else
   puts "I can't guess the number"
end
```

```
x is 1
```

Ruby *if* 修饰符

语法

```
code if condition
```

如果 *conditional* 为真，则执行 *code*。

实例

```
#!/usr/bin/ruby

$debug=1
print "debug\n" if $debug
```

这将产生以下结果：

```
debug
```

Ruby *unless* 语句

语法

```
unless conditional [then]
   code
[else
   code ]
end
```

如果 *conditional* 为假，则执行 *code*。如果 *conditional* 为真，则执行 else 子句中指定的 *code*。

实例

```
#!/usr/bin/ruby

x=1
unless x>2
   puts "x is less than 2"
 else
  puts "x is greater than 2"
end
```

这将产生以下结果：

```
x is less than 2
```

Ruby *unless* 修饰符

语法

```
code unless conditional
```

如果 *conditional* 为假，则执行 *code*。

实例

```
#!/usr/bin/ruby

$var =  1
print "1 -- Value is set\n" if $var
print "2 -- Value is set\n" unless $var

$var = false
print "3 -- Value is set\n" unless $var
```

这将产生以下结果：

```
1 -- Value is set
3 -- Value is set
```

Ruby *case* 语句

语法

```
case expression
[when expression [, expression ...] [then]
   code ]...
[else
   code ]
end
```

比较 case 所指定的 *expression*，当使用 === 运算符指定时，执行匹配的 when 子句的 *code*。

when 子句所指定的 *expression* 背当作左操作数。如果没有匹配的 when 子句，*case* 执行 *else* 子句的代码。

when 语句的表达式通过保留字 *then*、一个换行符或一个分号，来与代码分离开。

因此：

```
case expr0
when expr1, expr2
   stmt1
when expr3, expr4
   stmt2
else
   stmt3
end
```

基本上类似于：

```
_tmp = expr0
if expr1 === _tmp || expr2 === _tmp
   stmt1
elsif expr3 === _tmp || expr4 === _tmp
   stmt2
else
   stmt3
end
```

实例

```
#!/usr/bin/ruby

$age =  5
case $age
when 0 .. 2
    puts "baby"
when 3 .. 6
    puts "little child"
when 7 .. 12
    puts "child"
when 13 .. 18
    puts "youth"
else
    puts "adult"
end
```

这将产生以下结果：

```
little child
```

Ruby 循环

Ruby 中的循环用于执行相同的代码块若干次。本章节将详细介绍 Ruby 支持的所有循环语句。

Ruby *while* 语句

语法

```
while conditional [do]
   code
end
```

当 *conditional* 为真时，执行 *code*。*while* 循环的 *conditional* 通过保留字 *do*、一个换行符、反斜线 \ 或一个分号 ; ，来与 *code* 分离开。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5

while $i < $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby *while* 修饰符

语法

```
code while condition

或者

begin
  code
end while conditional
```

当 *conditional* 为真时，执行 *code*。

如果 *while* 修饰符跟在一个没有 *rescue* 或 ensure 子句的 *begin* 语句后面，*code* 会在 *conditional* 判断之前执行一次。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1
end while $i < $num
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby *until* 语句

```
until conditional [do]
   code
end
```

当 *conditional* 为假时，执行 *code*。*until* 语句的 *conditional* 通过保留字 *do*、一个换行符或一个分号，来与 *code* 分离开。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5

until $i > $num  do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby *until* 修饰符

语法

```
code until conditional

OR

begin
   code
end until conditional
```

当 *conditional* 为假时，执行 *code*。

如果 *until* 修饰符跟在一个没有 *rescue* 或 ensure 子句的 *begin* 语句后面，*code* 会在 *conditional* 判断之前执行一次。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1;
end until $i > $num
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby *for* 语句

语法

```
for variable [, variable ...] in expression [do]
   code
end
```

针对 *expression* 中的每个元素分别执行一次 *code*。

实例

```
#!/usr/bin/ruby

for i in 0..5
   puts "Value of local variable is #{i}"
end
```

在这里，我们已经定义了范围 0..5。语句 for i in 0..5 允许 i 的值从 0 到 5（包含 5）。这将产生以下结果：

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

for...in 循环几乎是完全等价于：

```
(expression).each do |variable[, variable...]| code end
```

但是，for 循环不会为局部变量创建一个新的作用域。for 循环的 *expression* 通过保留字 *do*、一个换行符或一个分号，来与 *code* 分离开。.

实例

```
#!/usr/bin/ruby

(0..5).each do |i|
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby *break* 语句

语法

```
break
```

终止最内部的循环。如果在块内调用，则终止相关块的方法（方法返回 nil）。

实例

```
#!/usr/bin/ruby

for i in 0..5
   if i > 2 then
      break
   end
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
```

Ruby *next* 语句

语法

```
next
```

跳到最内部循环的下一个迭代。如果在块内调用，则终止块的执行（*yield* 或调用返回 nil）。

实例

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      next
   end
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby *redo* 语句

语法

```
redo
```

重新开始最内部循环的该次迭代，不检查循环条件。如果在块内调用，则重新开始 *yield* 或 *call*。

实例

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      puts "Value of local variable is #{i}"
      redo
   end
end
```

这将产生以下结果，并会进入一个无限循环：

```
Value of local variable is 0
Value of local variable is 0
............................
```

Ruby *retry* 语句

语法

```
retry
```

如果 *retry* 出现在 begin 表达式的 rescue 子句中，则从 begin 主体的开头重新开始。

```
begin
   do_something # 抛出的异常
rescue
   # 处理错误
   retry  # 重新从 begin 开始
end
```

如果 retry 出现在迭代内、块内或者 for 表达式的主体内，则重新开始迭代调用。迭代的参数会重新评估。

```
for i in 1..5
   retry if some_condition # 重新从 i == 1 开始
end
```

实例

```
#!/usr/bin/ruby

for i in 1..5
   retry if  i > 2
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果，并会进入一个无限循环：

```
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
............................
```

Ruby 方法

Ruby 方法与其他编程语言中的函数类似。Ruby 方法用于捆绑一个或多个重复的语句到一个单元中。

方法名应以小写字母开头。如果您以大写字母作为方法名的开头，Ruby 可能会把它当作常量，从而导致不正确地解析调用。

方法应在调用之前定义，否则 Ruby 会产生未定义的方法调用异常。

语法

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
   expr..
end
```

所以，您可以定义一个简单的方法，如下所示：

```
def method_name
   expr..
end
```

您可以定义一个接受参数的方法，如下所示：

```
def method_name (var1, var2)
   expr..
end
```

您可以为参数设置默认值，如果方法调用时未传递必需的参数则使用默认值：

```
def method_name (var1=value1, var2=value2)
   expr..
end
```

当您要调用方法时，只需要使用方法名即可，如下所示：

```
method_name
```

但是，当您调用带参数的方法时，您在写方法名时还要带上参数，例如：

```
method_name 25, 30
```

使用带参数方法最大的缺点是调用方法时需要记住参数个数。例如，如果您向一个接受三个参数的方法只传递了两个参数，Ruby 会显示错误。

实例

```
#!/usr/bin/ruby

def test(a1="Ruby", a2="Perl")
   puts "The programming language is #{a1}"
   puts "The programming language is #{a2}"
end
test "C", "C++"
test
```

这将产生以下结果：

```
The programming language is C
The programming language is C++
The programming language is Ruby
The programming language is Perl
```

从方法返回值

Ruby 中的每个方法默认都会返回一个值。这个返回的值是最后一个语句的值。例如：

```
def test
   i = 100
   j = 10
   k = 0
end
```

在调用这个方法时，将返回最后一个声明的变量 k。

Ruby *return* 语句

Ruby 中的 *return* 语句用于从 Ruby 方法中返回一个或多个值。

语法

```
return [expr[`,' expr...]]
```

如果给出超过两个的表达式，包含这些值的数组将是返回值。如果未给出表达式，nil 将是返回值。

实例

```
return

OR

return 12

OR

return 1,2,3
```

看看下面的实例：

```
#!/usr/bin/ruby

def test
   i = 100
   j = 200
   k = 300
return i, j, k
end
var = test
puts var
```

这将产生以下结果：

```
100
200
300
```

可变数量的参数

假设您声明了一个带有两个参数的方法，当您调用该方法时，您同时还需要传递两个参数。

但是，Ruby 允许您声明参数数量可变的方法。让我们看看下面的实例：

```
#!/usr/bin/ruby

def sample (*test)
   puts "The number of parameters is #{test.length}"
   for i in 0...test.length
      puts "The parameters are #{test[i]}"
   end
end
sample "Zara", "6", "F"
sample "Mac", "36", "M", "MCA"
```

在这段代码中，您已经声明了一个方法 sample，接受一个参数 test。但是，这个参数是一个变量参数。这意味着参数可以带有不同数量的变量。所以上面的代码将产生下面的结果：

```
The number of parameters is 3
The parameters are Zara
The parameters are 6
The parameters are F
The number of parameters is 4
The parameters are Mac
The parameters are 36
The parameters are M
The parameters are MCA
```

类方法

当方法定义在类定义外部时，方法默认标记为 *private*。另一方面，定义在类定义中的方法默认标记为 public。方法默认的可见性和 *private* 标记可通过模块（Module）的 *public* 或 *private* 改变。

当你想要访问类的方法时，您首先需要实例化类。然后，使用对象，您可以访问类的任何成员。

Ruby 提供了一种不用实例化类即可访问方法的方式。让我们看看如何声明并访问类方法：

```
class Accounts
   def reading_charge
   end
   def Accounts.return_date
   end
end
```

我们已经知道方法 return_date 是如何声明的。它是通过在类名后跟着一个点号，点号后跟着方法名来声明的。您可以直接访问类方法，如下所示：

```
Accounts.return_date
```

如需访问该方法，您不需要创建类 Accounts 的对象。

Ruby *alias* 语句

这个语句用于为方法或全局变量起别名。别名不能在方法主体内定义。即使方法被重写，方法的别名也保持方法的当前定义。

为编号的全局变量（$1, $2,...）起别名是被禁止的。重写内置的全局变量可能会导致严重的问题。

语法

```
alias method-name method-name
alias global-variable-name global-variable-name
```

实例

```
alias foo bar
alias $MATCH $&
```

在这里，我们已经为 bar 定义了别名为 foo，为 $& 定义了别名为 $MATCH。

Ruby *undef* 语句

这个语句用于取消方法定义。*undef* 不能出现在方法主体内。

通过使用 *undef* 和 *alias*，类的接口可以从父类独立修改，但请注意，在自身内部方法调用时，它可能会破坏程序。

语法

```
undef method-name
```

实例

下面的实例取消名为 *bar*的方法定义：

```
undef bar
```

Ruby 块

您已经知道 Ruby 如何定义方法以及您如何调用方法。类似地，Ruby 有一个块的概念。

- 块由大量的代码组成。
- 您需要给块取个名称。
- 块中的代码总是包含在大括号 {} 内。
- 块总是从与其具有相同名称的函数调用。这意味着如果您的块名称为 *test*，那么您要使用函数 *test* 来调用这个块。
- 您可以使用 *yield* 语句来调用块。

语法

```
block_name{
   statement1
   statement2
   ..........
}
```

在这里，您将学到如何使用一个简单的 *yield* 语句来调用块。您也将学到如何使用带有参数的 *yield* 语句来调用块。在实例中，您将看到这两种类型的 *yield* 语句。

yield 语句

让我们看一个 yield 语句的实例：

```
#!/usr/bin/ruby

def test
   puts "You are in the method"
   yield
   puts "You are again back to the method"
   yield
end
test {puts "You are in the block"}
```

这将产生以下结果：

```
You are in the method
You are in the block
You are again back to the method
You are in the block
```

您也可以传递带有参数的 yield 语句。下面是一个实例：

```
#!/usr/bin/ruby

def test
   yield 5
   puts "You are in the method test"
   yield 100
end
test {|i| puts "You are in the block #{i}"}
```

这将产生以下结果：

```
You are in the block 5
You are in the method test
You are in the block 100
```

在这里，*yield* 语句后跟着参数。您甚至可以传递多个参数。在块中，您可以在两个竖线之间放置一个变量来接受参数。因此，在上面的代码中，yield 5 语句向 test 块传递值 5 作为参数。

现在，看下面的语句：

```
test {|i| puts "You are in the block #{i}"}
```

在这里，值 5 会在变量 i 中收到。现在，观察下面的 puts 语句：

```
puts "You are in the block #{i}"
```

这个 puts 语句的输出是：

```
You are in the block 5
```

如果您想要传递多个参数，那么 *yield* 语句如下所示：

```
yield a, b
```

此时，块如下所示：

```
test {|a, b| statement}
```

参数使用逗号分隔。

块和方法

您已经看到块和方法之间是如何相互关联的。您通常使用 yield 语句从与其具有相同名称的方法调用块。因此，代码如下所示：

```
#!/usr/bin/ruby

def test
  yield
end
test{ puts "Hello world"}
```

本实例是实现块的最简单的方式。您使用 *yield* 语句调用 test 块。

但是如果方法的最后一个参数前带有 &，那么您可以向该方法传递一个块，且这个块可被赋给最后一个参数。如果 * 和 & 同时出现在参数列表中，& 应放在后面。

```
#!/usr/bin/ruby

def test(&block)
   block.call
end
test { puts "Hello World!"}
```

这将产生以下结果：

```
Hello World!
```

BEGIN 和 END 块

每个 Ruby 源文件可以声明当文件被加载时要运行的代码块（BEGIN 块），以及程序完成执行后要运行的代码块（END 块）。

```
#!/usr/bin/ruby

BEGIN {
  # BEGIN block code
  puts "BEGIN code block"
}

END {
  # END block code
  puts "END code block"
}
  # MAIN block code
puts "MAIN code block"
```

一个程序可以包含多个 BEGIN 和 END 块。BEGIN 块按照它们出现的顺序执行。END 块按照它们出现的相反顺序执行。当执行时，上面的程序产生产生以下结果：

```
BEGIN code block
MAIN code block
END code block
```

Ruby 模块（Module）

模块（Module）是一种把方法、类和常量组合在一起的方式。模块（Module）为您提供了两大好处。

- 模块提供了一个命名空间和避免名字冲突。
- 模块实现了 *mixin* 装置。

模块（Module）定义了一个命名空间，相当于一个沙箱，在里边您的方法和常量不会与其他地方的方法常量冲突。

语法

```
module Identifier
   statement1
   statement2
   ...........
end
```

模块常量命名与类常量命名类似，以大写字母开头。方法定义看起来也相似：模块方法定义与类方法定义类似。

通过类方法，您可以在类方法名称前面放置模块名称和一个点号来调用模块方法，您可以使用模块名称和两个冒号来引用一个常量。

实例

```
#!/usr/bin/ruby

# 定义在 trig.rb 文件中的模块

module Trig
   PI = 3.141592654
   def Trig.sin(x)
   # ..
   end
   def Trig.cos(x)
   # ..
   end
end
```

我们可以定义多个函数名称相同但是功能不同的模块：

```
#!/usr/bin/ruby

# 定义在 moral.rb 文件中的模块

module Moral
   VERY_BAD = 0
   BAD = 1
   def Moral.sin(badness)
   # ...
   end
end
```

就像类方法，当您在模块中定义一个方法时，您可以指定在模块名称后跟着一个点号，点号后跟着方法名。

Ruby *require* 语句

require 语句类似于 C 和 C++ 中的 include 语句以及 Java 中的 import 语句。如果一个第三方的程序想要使用任何已定义的模块，则可以简单地使用 Ruby *require* 语句来加载模块文件：

语法

```
require filename
```

在这里，文件扩展名 **.rb** 不是必需的。

实例

```
$LOAD_PATH << '.'

require 'trig.rb'
require 'moral'

y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

在这里，我们使用 **$LOAD_PATH << '.'** 让 Ruby 知道必须在当前目录中搜索被引用的文件。如果您不想使用 $LOAD_PATH，那么您可以使用 **require_relative** 来从一个相对目录引用文件。

注意：在这里，文件包含相同的函数名称。所以，这会在引用调用程序时导致代码模糊，但是模块避免了这种代码模糊，而且我们可以使用模块的名称调用适当的函数。

Ruby *include* 语句

您可以在类中嵌入模块。为了在类中嵌入模块，您可以在类中使用 *include* 语句：

语法

```
include modulename
```

如果模块是定义在一个单独的文件中，那么在嵌入模块之前使用 *require* 语句引用该文件时必需的。

实例

假设下面的模块写在 *support.rb* 文件中。

```
module Week
   FIRST_DAY = "Sunday"
   def Week.weeks_in_month
      puts "You have four weeks in a month"
   end
   def Week.weeks_in_year
      puts "You have 52 weeks in a year"
   end
end
```

现在，您可以在类中引用该模块，如下所示：

```
#!/usr/bin/ruby
$LOAD_PATH << '.'
require "support"

class Decade
include Week
   no_of_yrs=10
   def no_of_months
      puts Week::FIRST_DAY
      number=10*12
      puts number
   end
end
d1=Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
Week.weeks_in_year
d1.no_of_months
```

这将产生以下结果：

```
Sunday
You have four weeks in a month
You have 52 weeks in a year
Sunday
120
```

Ruby 中的 Mixins

在阅读本节之前，您需要初步了解面向对象的概念。

当一个类可以从多个父类继承类的特性时，该类显示为多重继承。

Ruby 不直接支持多重继承，但是 Ruby 的模块（Module）有另一个神奇的功能。它几乎消除了多重继承的需要，提供了一种名为 *mixin* 的装置。

Mixins 向您提供了一种完美的为类添加功能的控制方式。但是，它们真正的强大在于当 mixin 中的代码开始与使用它的类中的代码交互时。

让我们看看下面的示例代码，深入了解 mixin：

```
module A
   def a1
   end
   def a2
   end
end
module B
   def b1
   end
   def b2
   end
end

class Sample
include A
include B
   def s1
   end
end

samp=Sample.new
samp.a1
samp.a2
samp.b1
samp.b2
samp.s1
```

模块 A 由方法 a1 和 a2 组成。模块 B 由方法 b1 和 b2 组成。类 Sample 包含了模块 A 和 B。类 Sample 可以访问所有四个方法，即 a1、a2、b1 和 b2。因此，您可以看到类 Sample 继承了两个模块。因此，您可以说类 Sample 显示了多重继承或 *mixin* 。

Ruby 字符串（String）

Ruby 中的 String 对象存储并操作一个或多个字节的任意序列，通常表示那些代表人类语言的字符。

最简单的字符串是括在单引号（单引号字符）内。在引号标记内的文本是字符串的值：

```
'This is a simple Ruby string literal'
```

如果您需要在单引号字符串内使用单引号字符，那么需要在单引号字符串使用反斜杠，这样 Ruby 解释器就不会认为这个单引号字符会终止字符串：

```
'Won\'t you read O\'Reilly\'s book?'
```

反斜杠也能转义另一个反斜杠，这样第二个反斜杠本身不会解释为转义字符。

以下是 Ruby 中字符串相关的特性。

表达式置换

表达式置换是一种使用 #{ 和 } 把任意 Ruby 表达式的值嵌入到字符串中的方式：

```
#!/usr/bin/ruby

x, y, z = 12, 36, 72
puts "The value of x is #{ x }."
puts "The sum of x and y is #{ x + y }."
puts "The average was #{ (x + y + z)/3 }."
```

这将产生以下结果：

```
The value of x is 12.
The sum of x and y is 48.
The average was 40.
```

一般的分隔字符串

通过一般的分隔字符串，您可以在以百分号字符（%）为前导的一对匹配的任意分隔字符（例如，!、 (、 {、 <，等等）内创建字符串。Q、 q 和 x 有特殊的意义。一般的分隔字符串可以是：

```
%{Ruby is fun.}  相当于 "Ruby is fun."
%Q{ Ruby is fun. } 相当于 " Ruby is fun. "
%q[Ruby is fun.]  相当于以单引号字符串
%x!ls! 相当于反勾号命令输出 `ls`
```

转义字符

下标列出了可使用反斜杠符号转义的转义字符或非打印字符。

注意：在一个双引号括起的字符串内，转义字符会被解释；在一个单引号括起的字符串内，转义字符会被保留。

反斜杠符号	十六进制字符	描述
\a	0x07	报警符
\b	0x08	退格键
\cx		Control-x
\C-x		Control-x
\e	0x1b	转义符
\f	0x0c	换页符
\M-\C-x		Meta-Control-x
\n	0x0a	换行符
\nnn		八进制表示法，其中 n 的范围为 0.7
\r	0x0d	回车符
\s	0x20	空格符
\t	0x09	制表符
\v	0x0b	垂直制表符
\x		字符 x
\xnn		十六进制表示法，其中 n 的范围为 0.9、 a.f 或 A.F

字符编码

Ruby 的默认字符集是 ASCII，字符可用单个字节表示。如果您使用 UTF-8 或其他现代的字符集，字符可能是用一个到四个字节表示。

您可以在程序开头使用 $KCODE 改变字符集，如下所示：

```
$KCODE = 'u'
```

下面是 $KCODE 可能的值。

编码	描述
a	ASCII （与 none 相同）。这是默认的。
e	EUC。
n	None （与 ASCII 相同）。
u	UTF-8。

字符串内建方法

我们需要有一个 String 对象的实例来调用 String 方法。下面是创建 String 对象实例的方式：

```
new [String.new(str="")]
```

这将返回一个包含 *str* 副本的新的字符串对象。现在，使用 *str* 对象，我们可以调用任意可用的实例方法。例如：

```
#!/usr/bin/ruby

myStr = String.new("THIS IS TEST")
foo = myStr.downcase

puts "#{foo}"
```

这将产生以下结果：

```
this is test
```

下面是公共的字符串方法（假设 str 是一个 String 对象）：

方法	描述
str % arg	使用格式规范格式化字符串。如果 arg 包含一个以上的替代，那么 arg 必须是一个数组。如需了解更多格式规范的信息，请查看"内核模块"下的 sprintf。
str * integer	返回一个包含 integer 个 str 的新的字符串。换句话说，str 被重复了 integer 次。
str + other_str	连接 other_str 到 str。
str << obj	连接一个对象到字符串。如果对象是范围为 0.255 之间的固定数字 Fixnum，则它会被转换为一个字符。把它与 concat 进行比较。
str <=> other_str	把 str 与 other_str 进行比较，返回 -1（小于）、0（等于）或 1（大于）。比较是区分大小写的。

str == obj	检查 str 和 obj 的相等性。如果 obj 不是字符串，则返回 false，如果 str <=> obj，则返回 true，返回 0。
str =~ obj	根据正则表达式模式 obj 匹配 str。返回匹配开始的位置，否则返回 false。
str =~ obj	根据正则表达式模式 obj 匹配 str。返回匹配开始的位置，否则返回 false。
str.capitalize	把字符串转换为大写字母显示。
str.capitalize!	与 capitalize 相同，但是 str 会发生变化并返回。
str.casecmp	不区分大小写的字符串比较。
str.center	居中字符串。
str.chomp	从字符串末尾移除记录分隔符（$/），通常是 \n。如果没有记录分隔符，则不进行任何操作。
str.chomp!	与 chomp 相同，但是 str 会发生变化并返回。
str.chop	移除 str 中的最后一个字符。
str.chop!	与 chop 相同，但是 str 会发生变化并返回。
str.concat(other_str)	连接 other_str 到 str。
str.count(str, ...)	给一个或多个字符集计数。如果有多个字符集，则给这些集合的交集计数。
str.crypt(other_str)	对 str 应用单向加密哈希。参数是两个字符长的字符串，每个字符的范围为 a.z、 A.Z、 0.9、 . 或 /。
str.delete(other_str, ...)	返回 str 的副本，参数交集中的所有字符会被删除。
str.delete!(other_str, ...)	与 delete 相同，但是 str 会发生变化并返回。
str.downcase	返回 str 的副本，所有的大写字母会被替换为小写字母。
str.downcase!	与 downcase 相同，但是 str 会发生变化并返回。
str.dump	返回 str 的版本，所有的非打印字符被替换为 \nnn 符号，所有的特殊字符被转义。
str.each(separator=$/) { |substr| block }	使用参数作为记录分隔符（默认是 $/）分隔 str，传递每个子字符串给被提供的块。
str.each_byte { |fixnum| block }	传递 str 的每个字节给 block，以字节的十进制表示法返回每个字节。
str.each_line(separator=$/) { |substr| block }	使用参数作为记录分隔符（默认是 $/）分隔 str，传递每个子字符串给被提供的 block。
str.empty?	如果 str 为空（即长度为 0），则返回 true。
str.eql?(other)	如果两个字符串有先攻的长度和内容，则这两个字符串相等。
str.gsub(pattern,	返回 str 的副本，pattern 的所有出现都替换为

str.gsub(pattern, replacement) [or] str.gsub(pattern) { |match| block }	replacement 或 block 的值。pattern 通常是一个正则表达式 Regexp；如果是一个字符串 String，则没有正则表达式元字符被解释（即，/\d/ 将匹配一个数字，但 '\d' 将匹配一个反斜杠后跟一个 'd'）。
str[fixnum] [or] str[fixnum,fixnum] [or] str[range] [or] str[regexp] [or] str[regexp, fixnum] [or] str[other_str]	使用下列的参数引用 str：参数为一个 Fixnum，则返回 fixnum 的字符编码；参数为两个 Fixnum，则返回一个从偏移（第一个 fixnum）开始截至到长度（第二个 fixnum）为止的子字符串；参数为 range，则返回该范围内的一个子字符串；参数为 regexp，则返回匹配字符串的部分；参数为带有 fixnum 的 regexp，则返回 fixnum 位置的匹配数据；参数为 other_str，则返回匹配 other_str 的子字符串。一个负数的 Fixnum 从字符串的末尾 -1 开始。
str[fixnum] = fixnum [or] str[fixnum] = new_str [or] str[fixnum, fixnum] = new_str [or] str[range] = aString [or] str[regexp] =new_str [or] str[regexp, fixnum] =new_str [or] str[other_str] = new_str]	替换整个字符串或部分字符串。与 slice! 同义。
str.gsub!(pattern, replacement) [or] str.gsub! (pattern) { |match| block }	执行 String#gsub 的替换，返回 str，如果没有替换被执行则返回 nil。
str.hash	返回一个基于字符串长度和内容的哈希。
str.hex	把 str 的前导字符当作十六进制数字的字符串（一个可选的符号和一个可选的 0x），并返回相对应的数字。如果错误则返回零。
str.include? other_str [or] str.include? fixnum	如果 str 包含给定的字符串或字符，则返回 true。
str.index(substring [, offset]) [or] str.index(fixnum [, offset]) [or] str.index(regexp [, offset])	返回给定子字符串、字符（fixnum）或模式（regexp）在 str 中第一次出现的索引。如果未找到则返回 nil。如果提供了第二个参数，则指定在字符串中开始搜索的位置。
str.insert(index, other_str)	在给定索引的字符前插入 other_str，修改 str。负值索引从字符串的末尾开始计数，并在给定字符后插入。其意图是在给定的索引处开始插入一个字符串。
str.inspect	返回 str 的可打印版本，带有转义的特殊字符。
str.intern [or] str.to_sym	返回与 str 相对应的符号，如果之前不存在，则创建符号。
str.length	返回 str 的长度。把它与 size 进行比较。
str.ljust(integer, padstr=' ')	如果 integer 大于 str 的长度，则返回长度为 integer 的新字符串，新字符串以 str 左对齐，并以 padstr 作为填

str.lstrip	返回 str 的副本，移除了前导的空格。
str.lstrip!	从 str 中移除前导的空格，如果没有变化则返回 nil。
str.match(pattern)	如果 pattern 不是正则表达是，则把 pattern 转换为正则表达式 Regexp，然后在 str 上调用它的匹配方法。
str.oct	把 str 的前导字符当作十进制数字的字符串（一个可选的符号），并返回相对应的数字。如果转换失败，则返回 0。
str.replace(other_str)	把 str 中的内容替换为 other_str 中的相对应的值。
str.reverse	返回一个新字符串，新字符串是 str 的倒序。
str.reverse!	逆转 str，str 会发生变化并返回。
str.rindex(substring [, fixnum]) [or] str.rindex(fixnum [, fixnum]) [or] str.rindex(regexp [, fixnum])	返回给定子字符串、字符（fixnum）或模式（regexp）在 str 中最后一次出现的索引。如果未找到则返回 nil。如果提供了第二个参数，则指定在字符串中结束搜索的位置。超出该点的字符将不被考虑。
str.rjust(integer, padstr=' ')	如果 integer 大于 str 的长度，则返回长度为 integer 的新字符串，新字符串以 str 右对齐，并以 padstr 作为填充。否则，返回 str。
str.rstrip	返回 str 的副本，移除了尾随的空格。
str.rstrip!	从 str 中移除尾随的空格，如果没有变化则返回 nil。
str.scan(pattern) [or] str.scan(pattern) { |match, ...| block }	两种形式匹配 pattern（可以是一个正则表达式 Regexp 或一个字符串 String）遍历 str。针对每个匹配，会生成一个结果，结果会添加到结果数组中或传递给 block。如果 pattern 不包含分组，则每个独立的结果由匹配的字符串、>< 组成。如果 pattern 包含分组，每个独立的结果是一个包含每个分组入口的数组。
str.slice(fixnum) [or] str.slice(fixnum, fixnum) [or] str.slice(range) [or] str.slice(regexp) [or] str.slice(regexp, fixnum) [or] str.slice(other_str) See str[fixnum], etc. str.slice!(fixnum) [or] str.slice!(fixnum, fixnum) [or] str.slice!(range) [or] str.slice!(regexp) [or] str.slice!(other_str)	从 str 中删除指定的部分，并返回删除的部分。如果值超出范围，参数带有 Fixnum 的形式，将生成一个 IndexError。参数为 range 的形式，将生成一个 RangeError，参数为 Regexp 和 String 的形式，将忽略执行动作。
	基于分隔符，把 str 分成子字符串，并返回这些子字符串的数组。 如果 *pattern* 是一个字符串 String，那么在分割 str 时，它将作为分隔符使用。如果 pattern 是一个单一的空格，那么 str 是基于空格进行分割，会忽略前导空

str.split(pattern=$;, [limit])	一的空格，那么 str 是基于空格进行分割，会忽略前导空格和连续空格字符。 如果 *pattern* 是一个正则表达式 Regexp，则 str 在 pattern 匹配的地方被分割。当 pattern 匹配一个玲长度的字符串时，str 被分割成单个字符。 如果省略了 *pattern* 参数，则使用 $; 的值。如果 $; 为 nil（默认的），str 基于空格进行分割，就像是指定了 作为分隔符一样。 如果省略了 *limit* 参数，会抑制尾随的 null 字段。如果 limit 是一个正数，则最多返回该数量的字段（如果 limit 为 1，则返回整个字符串作为数组中的唯一入口）。如果 limit 是一个负数，则返回的字段数量不限制，且不抑制尾随的 null 字段。
str.squeeze([other_str]*)	使用为 String#count 描述的程序从 other_str 参数建立一系列字符。返回一个新的字符串，其中集合中出现的相同的字符会被替换为单个字符。如果没有给出参数，则所有相同的字符都被替换为单个字符。
str.squeeze!([other_str]*)	与 squeeze 相同，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.strip	返回 str 的副本，移除了前导的空格和尾随的空格。
str.strip!	从 str 中移除前导的空格和尾随的空格，如果没有变化则返回 nil。
str.sub(pattern, replacement) [or] str.sub(pattern) { |match| block }	返回 str 的副本，pattern 的第一次出现会被替换为 replacement 或 block 的值。pattern 通常是一个正则表达式 Regexp；如果是一个字符串 String，则没有正则表达式元字符被解释。
str.sub!(pattern, replacement) [or] str.sub!(pattern) { |match| block }	执行 String#sub 替换，并返回 str，如果没有替换执行，则返回 nil。
str.succ [or] str.next	返回 str 的继承。
str.succ! [or] str.next!	相当于 String#succ，但是 str 会发生变化并返回。
str.sum(n=16)	返回 str 中字符的 n-bit 校验和，其中 n 是可选的 Fixnum 参数，默认为 16。结果是简单地把 str 中每个字符的二进制值的总和，以 2n - 1 为模。这不是一个特别好的校验和。
str.swapcase	返回 str 的副本，所有的大写字母转换为小写字母，所有的小写字母转换为大写字母。
str.swapcase!	相当于 String#swapcase，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.to_f	返回把 str 中的前导字符解释为浮点数的结果。超出有效数字的末尾的多余字符会被忽略。如果在 str 的开头没有有效数字，则返回 0.0。该方法不会生成异常。
str.to_i(base=10)	返回把 str 中的前导字符解释为整数基数（基数为 2、8、 10 或 16）的结果。超出有效数字的末尾的多余字符会被忽略。如果在 str 的开头没有有效数字，则返回 0。该方法不会生成异常。

str.to_s [or] str.to_str	返回接收的值。
str.tr(from_str, to_str)	返回 str 的副本，把 from_str 中的字符替换为 to_str 中相对应的字符。如果 to_str 比 from_str 短，那么它会以最后一个字符进行填充。两个字符串都可以使用 c1.c2 符号表示字符的范围。如果 from_str 以 ^ 开头，则表示除了所列出的字符以外的所有字符。
str.tr!(from_str, to_str)	相当于 String#tr，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.tr_s(from_str, to_str)	把 str 按照 String#tr 描述的规则进行处理，然后移除会影响翻译的重复字符。
str.tr_s!(from_str, to_str)	相当于 String#tr_s，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.unpack(format)	根据 format 字符串解码 str（可能包含二进制数据），返回被提取的每个值的数组。format 字符由一系列单字符指令组成。每个指令后可以跟着一个数字，表示重复该指令的次数。星号（*）将使用所有剩余的元素。指令 sSiIlL 每个后可能都跟着一个下划线（_），为指定类型使用底层平台的本地尺寸大小，否则使用独立于平台的一致的尺寸大小。format 字符串中的空格会被忽略。
str.upcase	返回 str 的副本，所有的小写字母会被替换为大写字母。操作是环境不敏感的，只有字符 a 到 z 会受影响。
str.upcase!	改变 str 的内容为大写，如果没有变化则返回 nil。
str.upto(other_str) { |s| block }	遍历连续值，以 str 开始，以 other_str 结束（包含），轮流传递每个值给 block。String#succ 方法用于生成每个值。

字符串 unpack 指令 下表列出了方法 String#unpack 的解压指令。

指令	返回	描述
A	String	移除尾随的 null 和空格。
a	String	字符串。
B	String	从每个字符中提取位（首先是最高有效位）。
b	String	从每个字符中提取位（首先是最低有效位）。
C	Fixnum	提取一个字符作为无符号整数。
c	Fixnum	提取一个字符作为整数。
D, d	Float	把 sizeof(double) 长度的字符当作原生的 double。

E	Float	把 sizeof(double) 长度的字符当作 littleendian 字节顺序的 double。
e	Float	把 sizeof(float) 长度的字符当作 littleendian 字节顺序的 float。
F, f	Float	把 sizeof(float) 长度的字符当作原生的 float。
G	Float	把 sizeof(double) 长度的字符当作 network 字节顺序的 double。
g	Float	把 sizeof(float) 长度的字符当作 network 字节顺序的 float。
H	String	从每个字符中提取十六进制（首先是最高有效位）。
h	String	从每个字符中提取十六进制（首先是最低有效位）。
I	Integer	把 sizeof(int) 长度（通过 _ 修改）的连续字符当作原生的 integer。
i	Integer	把 sizeof(int) 长度（通过 _ 修改）的连续字符当作有符号的原生的 integer。
L	Integer	把四个（通过 _ 修改）连续字符当作无符号的原生的 long integer。
l	Integer	把四个（通过 _ 修改）连续字符当作有符号的原生的 long integer。
M	String	引用可打印的。
m	String	Base64 编码。
N	Integer	把四个字符当作 network 字节顺序的无符号的 long。
n	Fixnum	把两个字符当作 network 字节顺序的无符号的 short。
P	String	把 sizeof(char *) 长度的字符当作指针，并从引用的位置返回 \emph{len} 字符。
p	String	把 sizeof(char *) 长度的字符当作一个空结束字符的指针。
Q	Integer	把八个字符当作无符号的 quad word（64 位）。
q	Integer	把八个字符当作有符号的 quad word（64 位）。
S	Fixnum	把两个（如果使用 _ 则不同）连续字符当作 native 字节顺序的无符号的 short。
s	Fixnum	把两个（如果使用 _ 则不同）连续字符当作 native 字节顺序的有符号的 short。
U	Integer	UTF-8 字符，作为无符号整数。
u	String	UU 编码。
V	Fixnum	把四个字符当作 little-endian 字节顺序的无符号的 long。
v	Fixnum	把两个字符当作 little-endian 字节顺序的无符号的 short。
w	Integer	BER 压缩的整数。
X		向后跳过一个字符。
x		向前跳过一个字符。
Z	String	和 * 一起使用，移除尾随的 null 直到第一个 null。

@		跳过 length 参数给定的偏移量。

实例

尝试下面的实例，解压各种数据。

```
"abc \0\0abc \0\0".unpack('A6Z6')   #=> ["abc", "abc "]
"abc \0\0".unpack('a3a3')           #=> ["abc", " \000\000"]
"abc \0abc \0".unpack('Z*Z*')       #=> ["abc ", "abc "]
"aa".unpack('b8B8')                 #=> ["10000110", "01100001"]
"aaa".unpack('h2H2c')               #=> ["16", "61", 97]
"\xfe\xff\xfe\xff".unpack('sS')     #=> [-2, 65534]
"now=20is".unpack('M*')             #=> ["now is"]
"whole".unpack('xax2aX2aX1aX2a')    #=> ["h", "e", "l", "l", "o"]
```

Ruby 数组（Array）

Ruby 数组是任何对象的有序的、整数索引的集合。数组中的每个元素都与一个索引相关，并可通过索引进行获取。

数组的索引从 0 开始，这与 C 或 Java 中一样。一个负数的索引时相对于数组的末尾计数的，也就是说，索引为 -1 表示数组的最后一个元素，-2 表示数组中的倒数第二个元素，依此类推。

Ruby 数组可存储诸如 String、 Integer、 Fixnum、 Hash、 Symbol 等对象，甚至可以是其他 Array 对象。Ruby 数组不像其他语言中的数组那么刚性。当向数组添加元素时，Ruby 数组会自动增长。

创建数组

有多种方式创建或初始化数组。一种方式是通过 *new* 类方法：

```
names = Array.new
```

您可以在创建数组的同时设置数组的大小：

```
names = Array.new(20)
```

数组 *names* 的大小或长度为 20 个元素。您可以使用 size 或 length 方法返回数组的大小：

```
#!/usr/bin/ruby

names = Array.new(20)
puts names.size  # 返回 20
puts names.length # 返回 20
```

这将产生以下结果：

```
20
20
```

您可以给数组中的每个元素赋值，如下所示：

```
#!/usr/bin/ruby

names = Array.new(4, "mac")

puts "#{names}"
```

这将产生以下结果：

```
macmacmacmac
```

您也可以使用带有 new 的块，每个元素使用块中的计算结果来填充：

```
#!/usr/bin/ruby

nums = Array.new(10) { |e| e = e * 2 }

puts "#{nums}"
```

这将产生以下结果：

```
024681012141618
```

数组还有另一种方法，[]，如下所示：

```
nums = Array.[](1, 2, 3, 4,5)
```

数组创建的另一种形式如下所示：

```
nums = Array[1, 2, 3, 4,5]
```

在核心 Ruby 中可用的 *Kernel* 模块有一个 Array 方法，只接受单个参数。在这里，该方法带有一个范围作为参数来创建一个数字数组：

```
#!/usr/bin/ruby

digits = Array(0..9)

puts "#{digits}"
```

这将产生以下结果：

```
0123456789
```

数组内建方法

我们需要有一个 Array 对象的实例来调用 Array 方法。下面是创建 Array 对象实例的方式：

```
Array.[](...) [or] Array[...] [or] [...]
```

这将返回一个使用给定对象进行填充的新的数组。现在，使用创建的对象，我们可以调用任意可用的实例方法。例如：

```
#!/usr/bin/ruby

digits = Array(0..9)

num = digits.at(6)

puts "#{num}"
```

这将产生以下结果：

```
6
```

下面是公共的数组方法（假设 *array* 是一个 Array 对象）：

方法	描述
array & other_array	返回一个新的数组，包含两个数组中共同的元素，没有重复。
array *int [or] array* str	返回一个新的数组，新数组通过连接 self 的 int 副本创建的。带有 String 参数时，相当于 self.join(str)。
array + other_array	返回一个新的数组，新数组通过连接两个数组产生第三个数组创建的。
array - other_array	返回一个新的数组，新数组是从初始数组中移除了在 other_array 中出现的项的副本。
str <=> other_str	把 str 与 other_str 进行比较，返回 -1（小于）、0（等于）或 1（大于）。比较是区分大小写的。
array | other_array	通过把 other_array 加入 array 中，移除重复项，返回一个新的数组。
array << obj	把给定的对象附加到数组的末尾。该表达式返回数组本身，所以几个附加可以连在一起。
array <=> other_array	如果数组小于、等于或大于 other_array，则返回一个整数（-1、 0 或 +1）。
array == other_array	如果两个数组包含相同的元素个数，且每个元素与另一个数组中相对应的元素相等（根据 Object.==），那么这两个数组相等。
array[index] [or] array[start, length] [or] array[range] [or] array.slice(index) [or] array.slice(start, length) [or] array.slice(range)	返回索引为 *index* 的元素，或者返回从 *start* 开始直至 *length* 个元素的子数组，或者返回 *range* 指定的子数组。负值索引从数组末尾开始计数（-1 是最后一个元素）。如果 *index*（或开始索引）超出范围，则返回 *nil*。
array[index] = obj [or]	设置索引为 *index* 的元素，或者替换从 *start* 开始直至 *length* 个元素的子数组，或者替换 *range* 指定的子数

an_array or nil [or] array[range] = obj or an_array or nil	组。如果索引大于数组的当前容量，那么数组会自动增长。负值索引从数组末尾开始计数。如果 *length* 为零则插入元素。如果在第二种或第三种形式中使用了 *nil*，则从 *self* 删除元素。
array.abbrev(pattern = nil)	为 *self* 中的字符串计算明确的缩写集合。如果传递一个模式或一个字符串，只考虑当字符串匹配模式或者以该字符串开始时的情况。
array.assoc(obj)	搜索一个数组，其元素也是数组，使用 obj.== 把 obj 与每个包含的数组的第一个元素进行比较。如果匹配则返回第一个包含的数组，如果未找到匹配则返回 *nil*。
array.at(index)	返回索引为 index 的元素。一个负值索引从 *self* 的末尾开始计数。如果索引超出范围则返回 nil。
array.clear 从数组中移除所有的元素。	
array.collect { |item| block } [or] array.map { |item| block }	为 *self* 中的每个元素调用一次 *block*。创建一个新的数组，包含 block 返回的值。
array.collect! { |item| block } [or] array.map! { |item| block }	为 *self* 中的每个元素调用一次 *block*，把元素替换为 *block* 返回的值。
array.compact	返回 *self* 的副本，移除了所有的 *nil* 元素。
array.compact!	从数组中移除所有的 *nil* 元素。如果没有变化则返回 *nil*。
array.concat(other_array)	追加 other*array* 中的元素到 _*self* 中。
array.delete(obj) [or] array.delete(obj) { block }	从 *self* 中删除等于 *obj* 的项。如果未找到相等项，则返回 *nil*。如果未找到相等项且给出了可选的代码 *block*，则返回 *block* 的结果。
array.delete_at(index)	删除指定的 *index* 处的元素，并返回该元素。如果 index 超出范围，则返回 *nil*。
array.delete_if { |item| block }	当 *block* 为 true 时，删除 *self* 的每个元素。
array.each { |item| block }	为 *self* 中的每个元素调用一次 *block*，传递该元素作为参数。
array.each_index { |index| block }	与 Array#each 相同，但是传递元素的 *index*，而不是传递元素本身。
array.empty?	如果数组本身没有包含元素，则返回 true。
array.eql?(other)	如果 *array* 和 *other* 是相同的对象，或者两个数组带有相同的内容，则返回 true。
array.fetch(index) [or] array.fetch(index, default) [or] array.fetch(index) {	尝试返回位置 *index* 处的元素。如果 *index* 位于数组外部，则第一种形式会抛出 *IndexError* 异常，第二种形式会返回 *default*，第三种形式会返回调用 *block* 传入

|index| block }	*index* 的值。负值的 *index* 从数组末尾开始计数。
array.fill(obj) [or] array.fill(obj, start [, length]) [or] array.fill(obj, range) [or] array.fill { |index| block } [or] array.fill(start [, length]) { |index| block } [or] array.fill(range) { |index| block }	前面三种形式设置 *self* 的被选元素为 *obj*。以 *nil* 开头相当于零。*nil* 的长度相当于 *self.length*。最后三种形式用 block 的值填充数组。*block* 通过带有被填充的每个元素的绝对索引来传递。
array.first [or] array.first(n)	返回数组的第一个元素或前 *n* 个元素。如果数组为空，则第一种形式返回 *nil*，第二种形式返回一个空的数组。
array.flatten	返回一个新的数组，新数组是一个一维的扁平化的数组（递归）。
array.flatten!	把 *array* 进行扁平化。如果没有变化则返回 *nil*。（数组不包含子数组。）
array.frozen?	如果 *array* 被冻结（或排序时暂时冻结），则返回 true。
array.hash	计算数组的哈希代码。两个具有相同内容的数组将具有相同的哈希代码。
array.include?(obj)	如果 *self* 中包含 *obj*，则返回 true，否则返回 false。
array.index(obj)	返回 *self* 中第一个等于 obj 的对象的 *index*。如果未找到匹配则返回 *nil*。
array.indexes(i1, i2, ... iN) [or] array.indices(i1, i2, ... iN)	该方法在 Ruby 的最新版本中被废弃，所以请使用 Array#values_at。
array.indices(i1, i2, ... iN) [or] array.indexes(i1, i2, ... iN)	该方法在 Ruby 的最新版本中被废弃，所以请使用 Array#values_at。
array.insert(index, obj...)	在给定的 *index* 的元素前插入给定的值，index 可以是负值。
array.inspect	创建一个数组的可打印版本。
array.join(sep=$,)	返回一个字符串，通过把数组的每个元素转换为字符串，并使用 *sep* 分隔进行创建的。
array.last [or] array.last(n)	返回 *self* 的最后一个元素。如果数组为空，则第一种形式返回 *nil*。
array.length	返回 *self* 中元素的个数。可能为零。
array.map { |item| block } [or] array.collect { |item| block }	为 *self* 的每个元素调用一次 *block*。创建一个新的数组，包含 block 返回的值。
array.map! { |item| block }	

[or] array.collect! { |item| block }	block 返回的值。
array.nitems	返回 *self* 中 non-nil 元素的个数。可能为零。
array.pack(aTemplateString)	根据 aTemplateString 中的指令，把数组的内容压缩为二进制序列。指令 A、 a 和 Z 后可以跟一个表示结果字段宽度的数字。剩余的指令也可以带有一个表示要转换的数组元素个数的数字。如果数字是一个星号（*），则所有剩余的数组元素都将被转换。任何指令后都可以跟一个下划线（_），表示指定类型使用底层平台的本地尺寸大小，否则使用独立于平台的一致的尺寸大小。在模板字符串中空格会被忽略。
array.pop	从 *array* 中移除最后一个元素，并返回该元素。如果 *array* 为空则返回 *nil*。
array.push(obj, ...)	把给定的 obj 附加到数组的末尾。该表达式返回数组本身，所以几个附加可以连在一起。
array.rassoc(key)	搜索一个数组，其元素也是数组，使用 == 把 *key* 与每个包含的数组的第二个元素进行比较。如果匹配则返回第一个包含的数组。
array.reject { |item| block }	返回一个新的数组，包含当 block 不为 true 时的数组项。
array.reject! { |item| block }	当 block 为真时，从 *array* 删除元素，如果没有变化则返回 *nil*。相当于 Array#delete_if。
array.replace(other_array)	把 *array* 的内容替换为 *other_array* 的内容，必要的时候进行截断或扩充。
array.reverse	返回一个新的数组，包含倒序排列的数组元素。
array.reverse!	把 *array* 进行逆转。
array.reverse_each {|item| block }	与 Array#each 相同，但是把 *array* 进行逆转。
array.rindex(obj)	返回 array 中最后一个等于 obj 的对象的索引。如果未找到匹配，则返回 *nil*。
array.select {|item| block }	调用从数组传入连续元素的 block，返回一个数组，包含 block 返回 *true* 值时的元素。
array.shift	返回 *self* 的第一个元素，并移除该元素（把所有的其他元素下移一位）。如果数组为空，则返回 *nil*。
array.size	返回 *array* 的长度（元素的个数）。length 的别名。
array.slice(index) [or] array.slice(start, length) [or] array.slice(range) [or] array[index] [or] array[start, length] [or] array[range]	返回索引为 *index* 的元素，或者返回从 *start* 开始直至 *length* 个元素的子数组，或者返回 *range* 指定的子数组。负值索引从数组末尾开始计数（-1 是最后一个元素）。如果 *index*（或开始索引）超出范围，则返回 *nil*。
array.slice!(index) [or]	删除 *index*（长度是可选的）或 *range* 指定的元素。返

array.slice!(start, length) [or] array.slice!(range)	回被删除的对象、子数组，如果 *index* 超出范围，则返回 *nil*。
array.sort [or] array.sort { | a,b | block }	返回一个排序的数组。
array.sort! [or] array.sort! { | a,b | block }	把数组进行排序。
array.to_a	返回 *self*。如果在 *Array* 的子类上调用，则把接收参数转换为一个 Array 对象。
array.to_ary	返回 self。
array.to_s	返回 self.join。
array.transpose	假设 self 是数组的数组，且置换行和列。
array.uniq	返回一个新的数组，移除了 *array* 中的重复值。
array.uniq!	从 *self* 中移除重复元素。如果没有变化（也就是说，未找到重复），则返回 *nil*。
array.unshift(obj, ...)	把对象前置在数组的前面，其他元素上移一位。
array.values_at(selector,...)	返回一个数组，包含 self 中与给定的 *selector*（一个或多个）相对应的元素。选择器可以是整数索引或者范围。
array.zip(arg, ...) [or] array.zip(arg, ...){ | arr | block }	把任何参数转换为数组，然后把 *array* 的元素与每个参数中相对应的元素合并。

数组 pack 指令

下表列出了方法 Array#pack 的压缩指令。

指令	描述
@	移动到绝对位置。
A	ASCII 字符串（填充 space，count 是宽度）。
a	ASCII 字符串（填充 null，count 是宽度）。
B	位字符串（降序）
b	位字符串（升序）。
C	无符号字符。
c	字符。
D, d	双精度浮点数，原生格式。
E	双精度浮点数，little-endian 字节顺序。

e	单精度浮点数，little-endian 字节顺序。
F, f	单精度浮点数，原生格式。
G	双精度浮点数，network（big-endian）字节顺序。
g	单精度浮点数，network（big-endian）字节顺序。
H	十六进制字符串（高位优先）。
h	十六进制字符串（低位优先）。
I	无符号整数。
i	整数。
L	无符号 long。
l	Long。
M	引用可打印的，MIME 编码。
m	Base64 编码字符串。
N	Long，network（big-endian）字节顺序。
n	Short，network（big-endian）字节顺序。
P	指向一个结构（固定长度的字符串）。
p	指向一个空结束字符串。
Q, q	64 位数字。
S	无符号 short。
s	Short。
U	UTF-8。
u	UU 编码字符串。
V	Long，little-endian 字节顺序。
v	Short，little-endian 字节顺序。
w	BER 压缩的整数 \fnm。
X	向后跳过一个字节。
x	Null 字节。
Z	与 a 相同，除了 null 会被加上 *。

实例

尝试下面的实例，压缩各种数据。

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
puts a.pack("A3A3A3")   #=> "a  b  c  "
puts a.pack("a3a3a3")   #=> "a\000\000b\000\000c\000\000"
puts n.pack("ccc")      #=> "ABC"
```

这将产生以下结果：

```
a  b  c
abc
ABC
```

Ruby 哈希（Hash）

哈希（Hash）是类似 "employee" => "salary" 这样的键值对的集合。哈希的索引是通过任何对象类型的任意键来完成的，而不是一个整数索引，其他与数组相似。

通过键或值遍历哈希的顺序看起来是随意的，且通常不是按照插入顺序。如果您尝试通过一个不存在的键访问哈希，则方法会返回 *nil*。

创建哈希

与数组一样，有各种不同的方式来创建哈希。您可以通过 *new* 类方法创建一个空的哈希：

```
months = Hash.new
```

您也可以使用 *new* 创建带有默认值的哈希，不带默认值的哈希是 *nil*：

```
months = Hash.new( "month" )

or

months = Hash.new "month"
```

当您访问带有默认值的哈希中的任意键时，如果键或值不存在，访问哈希将返回默认值：

```
#!/usr/bin/ruby

months = Hash.new( "month" )

puts "#{months[0]}"
puts "#{months[72]}"
```

这将产生以下结果：

```
month
month
```

```
#!/usr/bin/ruby

H = Hash["a" => 100, "b" => 200]

puts "#{H['a']}"
puts "#{H['b']}"
```

这将产生以下结果：

```
100
200
```

您可以使用任何的 Ruby 对象作为键或值，甚至可以使用数组，所以下面的实例是一个有效的实例：

```
[1,"jan"] => "January"
```

哈希内建方法

我们需要有一个 Hash 对象的实例来调用 Hash 方法。下面是创建 Hash 对象实例的方式：

```
Hash[[key =>|, value]* ] or

Hash.new [or] Hash.new(obj) [or]

Hash.new { |hash, key| block }
```

这将返回一个使用给定对象进行填充的新的哈希。现在，使用创建的对象，我们可以调用任意可用的实例方法。例如：

```
#!/usr/bin/ruby

$, = ", "
months = Hash.new( "month" )

months = {"1" => "January", "2" => "February"}

keys = months.keys

puts "#{keys}"
```

这将产生以下结果：

```
2, 1
```

下面是公共的哈希方法（假设 *hash* 是一个 Hash 对象）：

方法	描述
hash == other_hash	检查两个哈希是否具有相同的键值对个数，键值对是否相互匹配，来判断两个哈希是否相等。
hash.[key]	使用键，从哈希引用值。如果未找到键，则返回默认值。
hash.[key]=value	把 *value* 给定的值与 *key* 给定的键进行关联。
hash.clear	从哈希中移除所有的键值对。
	返回 *hash* 的默认值，如果未通过 default= 进行设置，则

hash.default(key = nil)	返回 nil。（如果键在 *hash* 中不存在，则 [] 返回一个默认值。）
hash.default = obj	为 *hash* 设置默认值。
hash.default_proc	如果 *hash* 通过块来创建，则返回块。
hash.delete(key) [or] array.delete(key) { |key| block }	通过 *key* 从 *hash* 中删除键值对。如果使用了块 且未找到匹配的键值对，则返回块的结果。把它与 *delete_if* 进行比较。
hash.delete_if { |key,value| block }	为 block 为 *true* 的每个块，从 *hash* 中删除键值对。
hash.each { |key,value| block }	遍历 *hash*，为每个 *key* 调用一次 block，传递 key-value 作为一个二元素数组。
hash.each_key { |key| block }	遍历 *hash*，为每个 *key* 调用一次 block，传递 *key* 作为参数。
hash.each_key { |key_value_array| block }	遍历 *hash*，为每个 *key* 调用一次 block，传递 *key* 和 *value* 作为参数。
hash.each_key { |value| block }	遍历 *hash*，为每个 *key* 调用一次 block，传递 *value* 作为参数。
hash.empty?	检查 hash 是否为空（不包含键值对），返回 *true* 或 *false*。
hash.fetch(key [, default]) [or] hash.fetch(key) { | key | block }	通过给定的 *key* 从 *hash* 返回值。如果未找到 *key*，且未提供其他参数，则抛出 *IndexError* 异常；如果给出了 *default*，则返回 *default*；如果指定了可选的 block，则返回 block 的结果。
hash.has_key?(key) [or] hash.include?(key) [or] hash.key?(key) [or] hash.member?(key)	检查给定的 *key* 是否存在于哈希中，返回 *true* 或 *false*。
hash.has_value?(value)	检查哈希是否包含给定的 *value*。
hash.index(value)	为给定的 *value* 返回哈希中的 *key*，如果未找到匹配值则返回 *nil*。
hash.indexes(keys)	返回一个新的数组，由给定的键的值组成。找不到的键将插入默认值。该方法已被废弃，请使用 select。
hash.indices(keys)	返回一个新的数组，由给定的键的值组成。找不到的键将插入默认值。该方法已被废弃，请使用 select。
hash.inspect	返回哈希的打印字符串版本。
hash.invert	创建一个新的 *hash*，倒置 *hash* 中的 *keys* 和 *values*。也就是说，在新的哈希中，*hash* 中的键将变成值，值将变成键。
hash.keys	创建一个新的数组，带有 *hash* 中的键。/td>
hash.length	以整数形式返回 *hash* 的大小或长度。

hash.merge(other_hash) [or] hash.merge(other_hash) { |key, oldval, newval| block }	返回一个新的哈希，包含 *hash* 和 *other_hash* 的内容，重写 hash 中与 *other_hash* 带有重复键的键值对。
hash.merge!(other_hash) [or] hash.merge!(other_hash) { |key, oldval, newval| block }	与 merge 相同，但实际上 hash 发生了变化。
hash.rehash	基于每个 *key* 的当前值重新建立 *hash*。如果插入后值发生了改变，该方法会重新索引 *hash*。
hash.reject { |key, value| block }	为 *block* 为 *true* 的每个键值对创建一个新的 *hash*。
hash.reject! { |key, value| block }	与 *reject* 相同，但实际上 hash 发生了变化。
hash.replace(other_hash)	把 *hash* 的内容替换为 *other_hash* 的内容。
hash.select { |key, value| block }	返回一个新的数组，由 *block* 返回 *true* 的 *hash* 中的键值对组成。
hash.shift	从 *hash* 中移除一个键值对，并把该键值对作为二元素数组返回。
hash.size	以整数形式返回 *hash* 的 *size* 或 length。
hash.sort	把 *hash* 转换为一个包含键值对数组的二维数组，然后进行排序。
hash.store(key, value)	存储 *hash* 中的一个键值对。
hash.to_a	从 hash 中创建一个二维数组。每个键值对转换为一个数组，所有这些数组都存储在一个数组中。
hash.to_hash	返回 *hash*（self）。
hash.to_s	把 *hash* 转换为一个数组，然后把该数组转换为一个字符串。
hash.update(other_hash) [or] hash.update(other_hash) {|key, oldval, newval| block}	返回一个新的哈希，包含 *hash* 和 *other_hash* 的内容，重写 *hash* 中与 *other_hash* 带有重复键的键值对。
hash.value?(value)	检查 *hash* 是否包含给定的 *value*。
hash.values	返回一个新的数组，包含 *hash* 的所有值。
hash.values_at(obj, ...)	返回一个新的数组，包含 *hash* 中与给定的键相关的值。

Ruby 日期 & 时间（Date & Time）

Time 类在 Ruby 中用于表示日期和时间。它是基于操作系统提供的系统日期和时间之上。该类可能无法表示 1970 年之前或者 2038 年之后的日期。

本教程将让您熟悉日期和时间的所有重要的概念。

创建当前的日期和时间

下面是获取当前的日期和时间的简单实例：

```
#!/usr/bin/ruby -w

time1 = Time.new

puts "Current Time : " + time1.inspect

# Time.now 是一个同义词
time2 = Time.now
puts "Current Time : " + time2.inspect
```

这将产生以下结果：

```
Current Time : Mon Jun 02 12:02:39 -0700 2008
Current Time : Mon Jun 02 12:02:39 -0700 2008
```

获取 Date & Time 组件

我们可以使用 *Time* 对象来获取各种日期和时间的组件。请看下面的实例：

```
#!/usr/bin/ruby -w

time = Time.new

# Time 的组件
puts "Current Time : " + time.inspect
puts time.year    # => 日期的年份
puts time.month   # => 日期的月份（1 到 12）
puts time.day     # => 一个月中的第几天（1 到 31）
puts time.wday    # => 一周中的星期几（0 是星期日）
puts time.yday    # => 365：一年中的第几天
puts time.hour    # => 23：24 小时制
puts time.min     # => 59
puts time.sec     # => 59
puts time.usec    # => 999999：微秒
puts time.zone    # => "UTC"：时区名称
```

这将产生以下结果：

```
Current Time : Mon Jun 02 12:03:08 -0700 2008
2008
6
2
1
154
12
3
8
247476
UTC
```

Time.utc、*Time.gm* 和 *Time.local* 函数

这些函数可用于格式化标准格式的日期，如下所示：

```
# July 8, 2008
Time.local(2008, 7, 8)
# July 8, 2008, 09:10am，本地时间
Time.local(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10 UTC
Time.utc(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10:11 GMT （与 UTC 相同）
Time.gm(2008, 7, 8, 9, 10, 11)
```

下面的实例在数组中获取所有的组件：

```
[sec,min,hour,day,month,year,wday,yday,isdst,zone]
```

尝试下面的实例：

```
#!/usr/bin/ruby -w

time = Time.new

values = time.to_a
p values
```

这将产生以下结果：

```
[26, 10, 12, 2, 6, 2008, 1, 154, false, "MST"]
```

该数组可被传到 *Time.utc* 或 *Time.local* 函数来获取日期的不同格式，如下所示：

```
#!/usr/bin/ruby -w

time = Time.new

values = time.to_a
puts Time.utc(*values)
```

这将产生以下结果：

```
Mon Jun 02 12:15:36 UTC 2008
```

下面是获取时间的方式，从纪元以来的秒数（平台相关）：

```
# 返回从纪元以来的秒数
time = Time.now.to_i

# 把秒数转换为 Time 对象
Time.at(time)

# 返回从纪元以来的秒数，包含微妙
time = Time.now.to_f
```

时区和夏令时

您可以使用 *Time* 对象来获取与时区和夏令时有关的所有信息，如下所示：

```
time = Time.new

# 这里是解释
time.zone        # => "UTC"：返回时区
time.utc_offset # => 0：UTC 是相对于 UTC 的 0 秒偏移
time.zone        # => "PST"（或其他时区）
time.isdst       # => false：如果 UTC 没有 DST（夏令时）
time.utc?        # => true：如果在 UTC 时区
time.localtime   # 转换为本地时区
time.gmtime      # 转换回 UTC
time.getlocal    # 返回本地区中的一个新的 Time 对象
time.getutc      # 返回 UTC 中的一个新的 Time 对象
```

格式化时间和日期

有多种方式格式化日期和时间。下面的实例演示了其中一部分：

```
#!/usr/bin/ruby -w
time = Time.new

puts time.to_s
puts time.ctime
puts time.localtime
puts time.strftime("%Y-%m-%d %H:%M:%S")
```

这将产生以下结果：

```
Mon Jun 02 12:35:19 -0700 2008
Mon Jun  2 12:35:19 2008
Mon Jun 02 12:35:19 -0700 2008
2008-06-02 12:35:19
```

时间格式化指令

下表所列出的指令与方法 *Time.strftime* 一起使用。

指令	描述
%a	星期几名称的缩写（比如 Sun）。
%A	星期几名称的全称（比如 Sunday）。
%b	月份名称的缩写（比如 Jan）。
%B	月份名称的全称（比如 January）。
%c	优选的本地日期和时间表示法。
%d	一个月中的第几天（01 到 31）。
%H	一天中的第几小时，24 小时制（00 到 23）。
%I	一天中的第几小时，12 小时制（01 到 12）。
%j	一年中的第几天（001 到 366）。
%m	一年中的第几月（01 到 12）。
%M	小时中的第几分钟（00 到 59）。
%p	子午线指示（AM 或 PM）。
%S	分钟中的第几秒（00 或 60）。
%U	当前年中的周数，从第一个星期日（作为第一周的第一天）开始（00 到 53）。
%W	当前年中的周数，从第一个星期一（作为第一周的第一天）开始（00 到 53）。
%w	一星期中的第几天（Sunday 是 0，0 到 6）。
%x	只有日期没有时间的优先表示法。
%X	只有时间没有日期的优先表示法。
%y	不带世纪的年份表示（00 到 99）。
%Y	带有世纪的年份。
%Z	时区名称。
%%	% 字符。

时间算法

您可以用时间做一些简单的算术，如下所示：

```
now = Time.now          # 当前时间
puts now

past = now - 10         # 10 秒之前。Time - number => Time
puts past

future = now + 10       # 从现在开始 10 秒之后。Time + number => Time
puts future

diff = future - now     # => 10  Time - Time => 秒数
puts diff
```

这将产生以下结果：

```
Thu Aug 01 20:57:05 -0700 2013
Thu Aug 01 20:56:55 -0700 2013
Thu Aug 01 20:57:15 -0700 2013
10.0
```

Ruby 范围（Range）

范围（Range）无处不在：January 到 December、 0 到 9、等等。Ruby 支持范围，并允许我们以不同的方式使用范围：

- 作为序列的范围
- 作为条件的范围
- 作为间隔的范围

作为序列的范围

范围的第一个也是最常见的用途是表达序列。序列有一个起点、一个终点和一个在序列产生连续值的方式。

Ruby 使用 **".."** 和 **"..."** 范围运算符创建这些序列。两点形式创建一个包含指定的最高值的范围，三点形式创建一个不包含指定的最高值的范围。

```
(1..5)        #==> 1, 2, 3, 4, 5
(1...5)       #==> 1, 2, 3, 4
('a'..'d')    #==> 'a', 'b', 'c', 'd'
```

序列 1..100 是一个 *Range* 对象，包含了两个 *Fixnum* 对象的引用。如果需要，您可以使用 *to_a* 方法把范围转换为列表。尝试下面的实例：

```
#!/usr/bin/ruby

$, =", "   # Array 值分隔符
range1 = (1..10).to_a
range2 = ('bar'..'bat').to_a

puts "#{range1}"
puts "#{range2}"
```

这将产生以下结果：

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
bar, bas, bat
```

范围实现了让您可以遍历它们的方法，您可以通过多种方式检查它们的内容：

```
#!/usr/bin/ruby

# Assume a range
digits = 0..9

puts digits.include?(5)
ret = digits.min
puts "Min value is #{ret}"

ret = digits.max
puts "Max value is #{ret}"

ret = digits.reject {|i| i < 5 }
puts "Rejected values are #{ret}"

digits.each do |digit|
   puts "In Loop #{digit}"
end
```

这将产生以下结果：

```
true
Min value is 0
Max value is 9
Rejected values are 5, 6, 7, 8, 9
In Loop 0
In Loop 1
In Loop 2
In Loop 3
In Loop 4
In Loop 5
In Loop 6
In Loop 7
In Loop 8
In Loop 9
```

作为条件的范围

范围也可以用作条件表达式。例如，下面的代码片段从标准输入打印行，其中每个集合的第一行包含单词 *start*，最后一行包含单词 *end.*：

```
while gets
   print if /start/../end/
end
```

范围可以用在 case 语句中：

```
#!/usr/bin/ruby

score = 70

result = case score
   when 0..40: "Fail"
   when 41..60: "Pass"
   when 61..70: "Pass with Merit"
   when 71..100: "Pass with Distinction"
   else "Invalid Score"
end

puts result
```

这将产生以下结果：

```
Pass with Merit
```

作为间隔的范围

范围的最后一个用途是间隔测试：检查某些值是否落在范围表示的间隔里。这是使用 === 相等运算符来完成的。

```
#!/usr/bin/ruby

if ((1..10) === 5)
  puts "5 lies in (1..10)"
end

if (('a'..'j') === 'c')
  puts "c lies in ('a'..'j')"
end

if (('a'..'j') === 'z')
  puts "z lies in ('a'..'j')"
end
```

这将产生以下结果：

```
5 lies in (1..10)
c lies in ('a'..'j')
```

Ruby 迭代器

迭代器是集合支持的方法。存储一组数据成员的对象称为集合。在 Ruby 中，数组和散列可以称之为集合。

迭代器返回集合的所有元素，一个接着一个。在这里我们将讨论两种迭代器，*each* 和 *collect*。

Ruby *each* 迭代器

each 迭代器返回数组或哈希的所有元素。

语法

```
collection.each do |variable|
   code
end
```

为集合中的每个元素执行 *code*。在这里，集合可以是数组或哈希。

实例

```
#!/usr/bin/ruby

ary = [1,2,3,4,5]
ary.each do |i|
   puts i
end
```

这将产生以下结果：

```
1
2
3
4
5
```

each 迭代器总是与一个块关联。它向块返回数组的每个值，一个接着一个。值被存储在变量 **i** 中，然后显示在屏幕上。

Ruby *collect* 迭代器

collect 迭代器返回集合的所有元素。

语法

```
collection = collection.collect
```

collect 方法不需要总是与一个块关联。*collect* 方法返回整个集合，不管它是数组或者是哈希。

实例

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = Array.new
b = a.collect
puts b
```

这将产生以下结果：

```
1
2
3
4
5
```

注意：*collect* 方法不是数组间进行复制的正确方式。这里有另一个称为 *clone* 的方法，用于复制一个数组到另一个数组。

当您想要对每个值进行一些操作以便获得新的数组时，您通常使用 collect 方法。例如，下面的代码会生成一个数组，其值是 a 中每个值的 10 倍。

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = a.collect{|x| 10*x}
puts b
```

这将产生以下结果：

```
10
20
30
40
50
```

Ruby 文件的输入与输出

Ruby 提供了一整套 I/O 相关的方法，在内核（Kernel）模块中实现。所有的 I/O 方法派生自 IO 类。

类 *IO* 提供了所有基础的方法，比如 *read*、 *write*、 *gets*、 *puts*、 *readline*、 *getc* 和 *printf*。

本章节将讲解所有 Ruby 中可用的基础的 I/O 函数。如需了解更多的函数，请查看 Ruby 的 *IO* 类。

puts 语句

在前面的章节中，您赋值给变量，然后使用 *puts* 语句打印输出。

puts 语句指示程序显示存储在变量中的值。这将在每行末尾添加一个新行。

实例

```
#!/usr/bin/ruby

val1 = "This is variable one"
val2 = "This is variable two"
puts val1
puts val2
```

这将产生以下结果：

```
This is variable one
This is variable two
```

gets 语句

gets 语句可用于获取来自名为 STDIN 的标准屏幕的用户输入。

实例

下面的代码演示了如何使用 gets 语句。该代码将提示用户输入一个值，该值将被存储在变量 val 中，最后会被打印在 STDOUT 上。

```
#!/usr/bin/ruby

puts "Enter a value :"
val = gets
puts val
```

这将产生以下结果：

```
Enter a value :
This is entered value
This is entered value
```

putc 语句

与 *puts* 语句不同，*puts* 语句输出整个字符串到屏幕上，而 *putc* 语句可用于依次输出一个字符。

实例

下面代码的输出只是字符 H：

```
#!/usr/bin/ruby

str="Hello Ruby!"
putc str
```

这将产生以下结果：

```
H
```

print 语句

print 语句与 *puts* 语句类似。唯一的不同在于 *puts* 语句在输出内容后会跳到下一行，而使用 *print* 语句时，光标定位在同一行。

实例

```
#!/usr/bin/ruby

print "Hello World"
print "Good Morning"
```

这将产生以下结果：

```
Hello WorldGood Morning
```

打开和关闭文件

截至现在，您已经读取并写入标准输入和输出。现在，我们将看看如何操作实际的数据文件。

File.new 方法

您可以使用 *File.new* 方法创建一个 *File* 对象用于读取、写入或者读写，读写权限取决于 mode 字符串。最后，您可以使用 *File.close* 方法来关闭该文件。

语法

```
aFile = File.new("filename", "mode")
   # ... 处理文件
aFile.close
```

File.open 方法

您可以使用 *File.open* 方法创建一个新的 file 对象，并把该 file 对象赋值给文件。但是，*File.open* 和 *File.new* 方法之间有一点不同。不同点是 *File.open* 方法可与块关联，而 *File.new* 方法不能。

```
File.open("filename", "mode") do |aFile|
   # ... process the file
end
```

下表列出了打开文件的不同模式：

模式	描述
r	只读模式。文件指针被放置在文件的开头。这是默认模式。
r+	读写模式。文件指针被放置在文件的开头。
w	只写模式。如果文件存在，则重写文件。如果文件不存在，则创建一个新文件用于写入。
w+	读写模式。如果文件存在，则重写已存在的文件。如果文件不存在，则创建一个新文件用于读写。
a	只写模式。如果文件存在，则文件指针被放置在文件的末尾。也就是说，文件是追加模式。如果文件不存在，则创建一个新文件用于写入。
a+	读写模式。如果文件存在，则文件指针被放置在文件的末尾。也就是说，文件是追加模式。如果文件不存在，则创建一个新文件用于读写。

读取和写入文件

用于简单 I/O 的方法也可用于所有 file 对象。所以，gets 从标准输入读取一行，*aFile.gets* 从文件对象 aFile 读取一行。

但是，I/O 对象提供了访问方法的附加设置，为我们提供了便利。

sysread 方法

您可以使用方法 *sysread* 来读取文件的内容。当使用方法 sysread 时，您可以使用任意一种模式打开文件。例如：

下面是输入文本文件：

```
This is a simple text file for testing purpose.
```

现在让我们尝试读取这个文件：

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r")
if aFile
   content = aFile.sysread(20)
   puts content
else
   puts "Unable to open file!"
end
```

该语句将输入文件的头 20 个字符。文件指针将被放置在文件中第 21 个字符的位置。

syswrite 方法

您可以使用方法 *syswrite* 来向文件写入内容。当使用方法 syswrite 时，您需要以写入模式打开文件。例如：

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
else
   puts "Unable to open file!"
end
```

该语句将写入 "ABCDEF" 到文件中。

each_byte 方法

该方法属于类 *File*。方法 *each_byte* 总是与块相关联。请看下面的代码实例：

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
   aFile.each_byte {|ch| putc ch; putc ?. }
else
   puts "Unable to open file!"
end
```

字符一个接着一个被传到变量 ch，然后显示在屏幕上，如下所示：

```
s. .a. .s.i.m.p.l.e. .t.e.x.t. .f.i.l.e. .f.o.r. .t.e.s.t.i.n.g. .p.u.r.p.o.s.e...
.
.
```

IO.readlines 方法

类 *File* 是类 IO 的一个子类。类 IO 也有一些用于操作文件的方法。

IO.readlines 是 IO 类中的一个方法。该方法逐行返回文件的内容。下面的代码显示了方法 *IO.readlines* 的使用：

```
#!/usr/bin/ruby

arr = IO.readlines("input.txt")
puts arr[0]
puts arr[1]
```

在这段代码中，变量 arr 是一个数组。文件 *input.txt* 的每一行将是数组 arr 中的一个元素。因此，arr[0] 将包含第一行，而 arr[1] 将包含文件的第二行。

IO.foreach 方法

该方法也逐行返回输出。方法 *foreach* 与方法 *readlines* 之间不同的是，方法 *foreach* 与块相关联。但是，不像方法 *readlines*，方法 *foreach* 不是返回一个数组。例如：

```
#!/usr/bin/ruby

IO.foreach("input.txt"){|block| puts block}
```

这段代码将把文件 *test* 的内容逐行传给变量 block，然后输出将显示在屏幕上。

重命名和删除文件

您可以通过 *rename* 和 *delete* 方法重命名和删除文件。

下面的实例重命名一个已存在文件 *test1.txt*：

```
#!/usr/bin/ruby

# 重命名文件 test1.txt 为 test2.txt
File.rename( "test1.txt", "test2.txt" )
```

下面的实例删除一个已存在文件 *test2.txt*：

```
#!/usr/bin/ruby

# 删除文件 test2.txt
File.delete("text2.txt")
```

文件模式与所有权

使用带有掩码的 *chmod* 方法来改变文件的模式或权限/访问列表：

下面的实例改变一个已存在文件 *test.txt* 的模式为一个掩码值：

```
#!/usr/bin/ruby

file = File.new( "test.txt", "w" )
file.chmod( 0755 )
```

下表列出了 *chmod* 方法中可使用的不同的掩码：

掩码	描述
0700	rwx 掩码，针对所有者
0400	r，针对所有者
0200	w，针对所有者
0100	x，针对所有者
0070	rwx 掩码，针对所属组
0040	r，针对所属组
0020	w，针对所属组
0010	x，针对所属组
0007	rwx 掩码，针对其他人
0004	r，针对其他人
0002	w，针对其他人
0001	x，针对其他人
4000	执行时设置用户 ID
2000	执行时设置所属组 ID
1000	保存交换文本，甚至在使用后也会保存

文件查询

下面的命令在打开文件前检查文件是否已存在：

```
#!/usr/bin/ruby

File.open("file.rb") if File::exists?( "file.rb" )
```

下面的命令查询文件是否确实是一个文件：

```
#!/usr/bin/ruby

# 返回 <i>true</i> 或 <i>false</i>
File.file?( "text.txt" )
```

下面的命令检查给定的文件名是否是一个目录：

```
#!/usr/bin/ruby

# 一个目录
File::directory?( "/usr/local/bin" ) # => true

# 一个文件
File::directory?( "file.rb" ) # => false
```

下面的命令检查文件是否可读、可写、可执行：

```
#!/usr/bin/ruby

File.readable?( "test.txt" )   # => true
File.writable?( "test.txt" )   # => true
File.executable?( "test.txt" ) # => false
```

下面的命令检查文件是否大小为零：

```
#!/usr/bin/ruby

File.zero?( "test.txt" )      # => true
```

下面的命令返回文件的大小：

```
#!/usr/bin/ruby

File.size?( "text.txt" )     # => 1002
```

下面的命令用于检查文件的类型：

```
#!/usr/bin/ruby

File::ftype( "test.txt" )     # => file
```

ftype 方法通过返回下列中的某个值来标识了文件的类型：*file*、 *directory*、 *characterSpecial*、 *blockSpecial*、 *fifo*、 *link*、 *socket* 或 *unknown*。

下面的命令用于检查文件被创建、修改或最后访问的时间：

```
#!/usr/bin/ruby

File::ctime( "test.txt" ) # => Fri May 09 10:06:37 -0700 2008
File::mtime( "text.txt" ) # => Fri May 09 10:44:44 -0700 2008
File::atime( "text.txt" ) # => Fri May 09 10:45:01 -0700 2008
```

Ruby 中的目录

所有的文件都是包含在目录中，Ruby 提供了处理文件和目录的方式。*File* 类用于处理文件，*Dir* 类用于处理目录。

浏览目录

为了在 Ruby 程序中改变目录，请使用 *Dir.chdir*。下面的实例改变当前目录为 */usr/bin*。

```
Dir.chdir("/usr/bin")
```

您可以通过 *Dir.pwd* 查看当前目录：

```
puts Dir.pwd # 返回当前目录，类似 /usr/bin
```

您可以使用 *Dir.entries* 获取指定目录内的文件和目录列表：

```
puts Dir.entries("/usr/bin").join(' ')
```

Dir.entries 返回一个数组，包含指定目录内的所有项。*Dir.foreach* 提供了相同的功能：

```
Dir.foreach("/usr/bin") do |entry|
   puts entry
end
```

获取目录列表的一个更简洁的方式是通过使用 Dir 的类数组的方法：

```
Dir["/usr/bin/*"]
```

创建目录

Dir.mkdir 可用于创建目录：

```
Dir.mkdir("mynewdir")
```

您也可以通过 mkdir 在新目录（不是已存在的目录）上设置权限：

注意：掩码 755 设置所有者（owner）、所属组（group）、每个人（world [anyone]）的权限为 rwxr-xr-x，其中 r = read 读取，w = write 写入，x = execute 执行。

```
Dir.mkdir( "mynewdir", 755 )
```

删除目录

Dir.delete 可用于删除目录。*Dir.unlink* 和 *Dir.rmdir* 执行同样的功能，为我们提供了便利。

```
Dir.delete("testdir")
```

创建文件 & 临时目录

临时文件是那些在程序执行过程中被简单地创建，但不会永久性存储的信息。

Dir.tmpdir 提供了当前系统上临时目录的路径，但是该方法默认情况下是不可用的。为了让 *Dir.tmpdir* 可用，使用必需的 'tmpdir' 是必要的。

您可以把 *Dir.tmpdir* 和 *File.join* 一起使用，来创建一个独立于平台的临时文件：

```
require 'tmpdir'
   tempfilename = File.join(Dir.tmpdir, "tingtong")
   tempfile = File.new(tempfilename, "w")
   tempfile.puts "This is a temporary file"
   tempfile.close
   File.delete(tempfilename)
```

这段代码创建了一个临时文件，并向其中写入数据，然后删除文件。Ruby 的标准库也包含了一个名为 *Tempfile* 的库，该库可用于创建临时文件：

```
require 'tempfile'
   f = Tempfile.new('tingtong')
   f.puts "Hello"
   puts f.path
   f.close
```

内建函数

下面提供了 Ruby 中处理文件和目录的内建函数的完整列表：

- File 类和方法。
- Dir 类和方法。

Ruby File 类和方法

File 表示一个连接到普通文件的 *stdio* 对象。open 为普通文件返回该类的一个实例。

类方法

方法	描述
File::atime(path)	返回 *path* 的最后访问时间。
File::basename(path[, suffix])	返回 *path* 末尾的文件名。如果指定了 *suffix*，则它会从文件名末尾被删除。 例如： File.basename("/home/users/bin/ruby.exe") #=> "ruby.exe"
File::blockdev?(path)	如果 path 是一个块设备，则返回 true。
File::chardev?(path)	如果 path 是一个字符设备，则返回 true。
File::chmod(mode, path...)	改变指定文件的权限模式。
File::chown(owner, group, path...)	改变指定文件的所有者和所属组。
File::ctime(path)	返回 path 的最后一个 inode 更改时间。
File::delete(path...) **File::unlink(path...)**	删除指定的文件。
File::directory?(path)	如果 path 是一个目录，则返回 true。
File::dirname(path)	返回 path 的目录部分，不包括最后的文件名。
File::executable?(path)	如果 path 是可执行的，则返回 true。
File::executable_real?(path)	如果 path 通过真正的用户权限是可执行的，则返回 true。
File::exist?(path)	如果 path 存在，则返回 true。
File::expand_path(path[, dir])	返回 path 的绝对路径，扩展 ~ 为进程所有者的主目录，~user 为用户的主目录。相对路径是相对于 dir 指定的目录，如果 dir 被省略则相对于当前工作目录。
File::file?(path)	如果 path 是一个普通文件，则返回 true。
File::ftype(path)	返回下列其中一个字符串，表示文件类型： **file** - 普通文件 **directory** - 目录 **characterSpecial** - 字符特殊文件 **blockSpecial** - 块特殊文件 **fifo** - 命名管道（FIFO） **link** - 符号链接 **socket** - Socket **unknown** - 未知的文件类型
File::grpowned?(path)	如果 path 由用户的所属组所有，则返回 true。

File::join(item...)	返回一个字符串，由指定的项连接在一起，并使用 File::Separator 进行分隔。 例如：File::join("", "home", "usrs", "bin") # => "/home/usrs/bin"
File::link(old, new)	创建一个到文件 old 的硬链接。
File::lstat(path)	与 stat 相同，但是它返回自身符号链接上的信息，而不是所指向的文件。
File::mtime(path)	返回 path 的最后一次修改时间。
File::new(path[, mode="r"]) File::open(path[, mode="r"]) File::open(path[, mode="r"]) {|f| ...}	打开文件。如果指定了块，则通过传递新文件作为参数来执行块。当块退出时，文件会自动关闭。这些方法有别于 Kernel.open，即使 path 是以 | 开头，后续的字符串也不会作为命令运行。
File::owned?(path)	如果 path 由有效的用户所有，则返回 true。
File::pipe?(path)	如果 path 是一个管道，则返回 true。
File::readable?(path)	如果 path 是可读的，则返回 true。
File::readable_real?(path)	如果 path 通过真正的用户权限是可读的，则返回 true。
File::readlink(path)	返回 path 所指向的文件。
File::rename(old, new)	改变文件名 old 为 new。
File::setgid?(path)	如果设置了 path 的 set-group-id 权限位，则返回 true。
File::setuid?(path)	如果设置了 path 的 set-user-id 权限位，则返回 true。
File::size(path)	返回 path 的文件大小。
File::size?(path)	返回 path 的文件大小，如果为 0 则返回 nil。
File::socket?(path)	如果 path 是一个 socket，则返回 true。
File::split(path)	返回一个数组，包含 path 的内容，path 被分成 File::dirname(path) 和 File::basename(path)。
File::stat(path)	返回 path 上带有信息的 File::Stat 对象。
File::sticky?(path)	如果设置了 path 的 sticky 位，则返回 true。
File::symlink(old, new)	创建一个指向文件 old 的符号链接。
File::symlink?(path)	如果 path 是一个符号链接，则返回 true。
File::truncate(path, len)	截断指定的文件为 len 字节。
File::unlink(path...)	删除 path 给定的文件。
File::umask([mask])	如果未指定参数，则为该进程返回当前的 umask。如果指定了一个参数，则设置了 umask，并返回旧的 umask。
File::utime(atime, mtime, path...)	改变指定文件的访问和修改时间。

File::writable?(path)	如果 path 是可写的，则返回 true。
File::writable_real?(path)	如果 path 通过真正的用户权限是可写的，则返回 true。
File::zero?(path)	如果 path 的文件大小是 0，则返回 true。

实例方法

假设 **f** 是 **File** 类的一个实例：

方法	描述
f.atime	返回 f 的最后访问时间。
f.chmode(mode)	改变 f 的权限模式。
f.chown(owner, group)	改变 f 的所有者和所属组。
f.ctime	返回 f 的最后一个 inode 更改时间。
f.flock(op)	调用 flock(2)。op 可以是 0 或一个逻辑值或 File 类常量 LOCK_EX、LOCK_NB、LOCK_SH 和 LOCK_UN。
f.lstat	与 stat 相同，但是它返回自身符号链接上的信息，而不是所指向的文件。
f.mtime	返回 f 的最后修改时间。
f.path	返回用于创建 f 的路径名。
f.reopen(path[, mode="r"])	重新打开文件。
f.truncate(len)	截断 f 为 len 字节。

Ruby Dir 类和方法

Dir | 是一个表示用于给出操作系统中目录中的文件名的目录流。Dir 类也拥有与目录相关的操作，比如通配符文件名匹配、改变工作目录等。

类方法

方法	描述
Dir[pat] Dir::glob(pat)	返回一个数组，包含与指定的通配符模式 pat 匹配的文件名： ***** - 匹配包含 null 字符串的任意字符串 ****** - 递归地匹配任意字符串 **?** - 匹配任意单个字符 **[...]** - 匹配封闭字符中的任意一个 **{a,b...}** - 匹配字符串中的任意一个 Dir["foo.*"] # 匹配 "foo.c"、 "foo.rb" 等等 Dir["foo.?"] # 匹配 "foo.c"、 "foo.h" 等等
Dir::chdir(path)	改变当前目录。
Dir::chroot(path)	改变根目录（只允许超级用户）。并不是在所有的平台上都可用。
Dir::delete(path)	删除 path 指定的目录。目录必须是空的。
Dir::entries(path)	返回一个数组，包含目录 path 中的文件名。
Dir::foreach(path) {| f| ...}	为 path 指定的目录中的每个文件执行一次块。
Dir::getwd Dir::pwd	返回当前目录。
Dir::mkdir(path[, mode=0777])	创建 path 指定的目录。权限模式可被 File::umask 的值修改，在 Win32 的平台上会被忽略。
Dir::new(path) Dir::open(path) Dir::open(path) {| dir| ...}	返回 path 的新目录对象。如果 open 给出一个块，则新目录对象会传到该块，块会在终止前关闭目录对象。
Dir::pwd	参见 Dir::getwd。
Dir::rmdir(path) Dir::unlink(path) Dir::delete(path)	删除 path 指定的目录。目录必须是空的。

实例方法

假设 **d** 是 **|Dir** 类的一个实例：

方法	描述
d.close	关闭目录流。
d.each {| f| ...}	为 d 中的每一个条目执行一次块。
d.pos	d.tell 返回 d 中的当前位置。
d.pos= offset	设置目录流中的位置。
d.pos= pos **d.seek(pos)**	移动到 d 中的某个位置。pos 必须是一个由 d.pos 返回的值或 0。
d.read	返回 d 的下一个条目。
d.rewind	移动 d 中的位置到第一个条目。
d.seek(po s)	参见 d.pos=pos。
d.tell	参见 d.pos。

Ruby 异常

异常和执行总是被联系在一起。如果您打开一个不存在的文件，且没有恰当地处理这种情况，那么您的程序则被认为是低质量的。

如果异常发生，则程序停止。异常用于处理各种类型的错误，这些错误可能在程序执行期间发生，所以要采取适当的行动，而不至于让程序完全停止。

Ruby 提供了一个完美的处理异常的机制。我们可以在 *begin/end* 块中附上可能抛出异常的代码，并使用 *rescue* 子句告诉 Ruby 完美要处理的异常类型。

语法

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# 其他异常
ensure
# 总是被执行
end
```

从 *begin* 到 *rescue* 中的一切是受保护的。如果代码块执行期间发生了异常，控制会传到 *rescue* 和 *end* 之间的块。

对于 *begin* 块中的每个 *rescue* 子句，Ruby 把抛出的异常与每个参数进行轮流比较。如果 rescue 子句中命名的异常与当前抛出的异常类型相同，或者是该异常的父类，则匹配成功。

如果异常不匹配所有指定的错误类型，我们可以在所有的 *rescue* 子句后使用一个 *else* 子句。

实例

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
      file = STDIN
end
print file, "==", STDIN, "\n"
```

这将产生以下结果。您可以看到，*STDIN* 取代了 *file*，因为打开失败。

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

使用 *retry* 语句

您可以使用 *rescue* 块捕获异常，然后使用 *retry* 语句从开头开始执行 *begin* 块。

语法

```
begin
    # 这段代码抛出的异常将被下面的 rescue 子句捕获
rescue
    # 这个块将捕获所有类型的异常
    retry  # 这将把控制移到 begin 的开头
end
```

实例

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
   fname = "existant_file"
   retry
end
```

以下是处理流程：

- 打开时发生异常。
- 跳到 rescue。fname 被重新赋值。
- 通过 retry 跳到 begin 的开头。
- 这次文件成功打开。
- 继续基本的过程。

注意：如果被重新命名的文件不存在，本势力代码会无限尝试。所以异常处理时，谨慎使用 *retry*。

使用 *raise* 语句

您可以使用 *raise* 语句抛出异常。下面的方法在调用时抛出异常。它的第二个消息将被输出。

语法

```
raise

OR

raise "Error Message"

OR

raise ExceptionType, "Error Message"

OR

raise ExceptionType, "Error Message" condition
```

第一种形式简单地重新抛出当前异常（如果没有当前异常则抛出一个 RuntimeError）。这用在传入异常之前需要解释异常的异常处理程序中。

第二种形式创建一个新的 *RuntimeError* 异常，设置它的消息为给定的字符串。该异常之后抛出到调用堆栈。

第三种形式使用第一个参数创建一个异常，然后设置相关的消息为第二个参数。

第四种形式与第三种形式类似，您可以添加任何额外的条件语句（比如 *unless*）来抛出异常。

实例

```
#!/usr/bin/ruby

begin
    puts 'I am before the raise.'
    raise 'An error has occurred.'
    puts 'I am after the raise.'
rescue
    puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

这将产生以下结果：

```
I am before the raise.
I am rescued.
I am after the begin block.
```

另一个演示 *raise* 用法的实例：

```
#!/usr/bin/ruby

begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
```

这将产生以下结果：

```
A test exception.
["main.rb:4"]
```

使用 *ensure* 语句

有时候，无论是否抛出异常，您需要保证一些处理在代码块结束时完成。例如，您可能在进入时打开了一个文件，当您退出块时，您需要确保关闭文件。

ensure 子句做的就是这个。ensure 放在最后一个 rescue 子句后，并包含一个块终止时总是执行的代码块。它与块是否正常退出、是否抛出并处理异常、是否因一个未捕获的异常而终止，这些都没关系，*ensure* 块始终都会运行。

语法

```
begin
   #.. 过程
   #.. 抛出异常
rescue
   #.. 处理错误
ensure
   #.. 最后确保执行
   #.. 这总是会执行
end
```

实例

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
ensure
  puts "Ensuring execution"
end
```

这将产生以下结果：

```
A test exception.
["main.rb:4"]
Ensuring execution
```

使用 *else* 语句

如果提供了 *else* 子句，它一般是放置在 *rescue* 子句之后，任意 *ensure* 之前。

else 子句的主体只有在代码主体没有抛出异常时执行。

语法

```
begin
   #.. 过程
   #.. 抛出异常
rescue
   #.. 处理错误
else
   #.. 如果没有异常则执行
ensure
   #.. 最后确保执行
   #.. 这总是会执行
end
```

实例

```
begin
 # 抛出 'A test exception.'
 puts "I'm not raising exception"
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
else
   puts "Congratulations-- no errors!"
ensure
  puts "Ensuring execution"
end
```

这将产生以下结果：

```
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

使用 $! 变量可以捕获抛出的错误消息。

Catch 和 Throw

raise 和 rescue 的异常机制能在发生错误时放弃执行，有时候需要在正常处理时跳出一些深层嵌套的结构。此时 catch 和 throw 就派上用场了。

catch 定义了一个使用给定的名称（可以是 Symbol 或 String）作为标签的块。块会正常执行知道遇到一个 throw。

语法

```
throw :lablename
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end

OR

throw :lablename condition
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end
```

实例

下面的实例中，如果用户键入 '!' 回应任何提示，使用一个 throw 终止与用户的交互。

```
def promptAndGet(prompt)
   print prompt
   res = readline.chomp
   throw :quitRequested if res == "!"
   return res
end

catch :quitRequested do
   name = promptAndGet("Name: ")
   age = promptAndGet("Age: ")
   sex = promptAndGet("Sex: ")
   # ..
   # 处理信息
end
promptAndGet("Name:")
```

上面的程序需要人工交互，您可以在您的计算机上进行尝试。这将产生以下结果：

```
Name: Ruby on Rails
Age: 3
Sex: !
Name:Just Ruby
```

类 Exception

Ruby 的标准类和模块抛出异常。所有的异常类组成一个层次，包括顶部的 Exception 类在内。下一层是七种不同的类型：

- Interrupt
- NoMemoryError
- SignalException
- ScriptError
- StandardError
- SystemExit

Fatal 是该层中另一种异常，但是 Ruby 解释器只在内部使用它。

ScriptError 和 StandardError 都有一些子类，但是在这里我们不需要了解这些细节。最重要的事情是创建我们自己的异常类，它们必须是类 Exception 或其子代的子类。

让我们看一个实例：

```
class FileSaveError < StandardError
   attr_reader :reason
   def initialize(reason)
      @reason = reason
   end
end
```

现在，看下面的实例，将用到上面的异常：

```
File.open(path, "w") do |file|
begin
    # 写出数据 ...
rescue
    # 发生错误
    raise FileSaveError.new($!)
end
end
```

在这里，最重要的一行是 raise *FileSaveError.new($!)*。我们调用 raise 来示意异常已经发生，把它传给 FileSaveError 的一个新的实例，由于特定的异常引起数据写入失败。

Ruby 高级

Ruby 面向对象

Ruby 是纯面向对象的语言，Ruby 中的一切都是以对象的形式出现。Ruby 中的每个值都是一个对象，即使是最原始的东西：字符串、数字，甚至连 true 和 false 都是对象。类本身也是一个对象，是 *Class* 类的一个实例。本章将向您讲解所有与 Ruby 面向对象相关的主要功能。

类用于指定对象的形式，它结合了数据表示法和方法，把数据整理成一个整齐的包。类中的数据和方法被称为类的成员。

Ruby 类定义

当您定义一个类时，您实际是定义了一个数据类型的蓝图。这实际上并没有定义任何的数据，而是定义了类的名称意味着什么，也就是说，定义了类的对象将由什么组成，以及在该对象上能执行什么操作。

类定义以关键字 **class** 开始，后跟类名称，最后以一个 **end** 进行分隔表示终止该类定义。例如，我们使用关键字 class 来定义 Box 类，如下所示：

```
class Box
   code
end
```

按照惯例，名称必须以大写字母开头，如果包含多个单词，每个单词首字母大写，但此间没有分隔符（例如：CamelCase）。

定义 Ruby 对象

类提供了对象的蓝图，所以基本上，对象是根据类进行创建的。我们使用 **new** 关键字声明类的对象。下面的语句声明了类 Box 的两个对象：

```
box1 = Box.new
box2 = Box.new
```

initialize 方法

initialize 方法是一个标准的 Ruby 类方法，与其他面向对象编程语言中的 **constructor** 工作原理类似。当您想要在创建对象的同时初始化一些类变量，initialize 方法就派上用场了。该方法带有一系列参数，与其他 Ruby 方法一样，使用该方法时，必须在前面放置 **def** 关键字，

如下所示：

```
class Box
   def initialize(w,h)
      @width, @height = w, h
   end
end
```

实例变量

实例变量是类属性，它们在使用类创建对象时就变成对象的属性。每个对象的属性是单独赋值的，和其他对象之间不共享值。在类的内部，是使用 @ 运算符访问这些属性，在类的外部，则是使用称为访问器方法的公共方法进行访问。下面我们以上面定义的类 **Box** 为实例，把 @width 和 @height 作为类 Box 的实例变量。

```
class Box
   def initialize(w,h)
      # 给实例变量赋值
      @width, @height = w, h
   end
end
```

访问器 & 设置器 方法

为了在类的外部使用变量，我们必须在访问器方法内部定义这些变量，这些访问器方法也被称为获取器方法。下面的实例演示了访问器方法的用法：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 访问器方法
   def printWidth
      @width
   end

   def printHeight
      @height
   end
end

# 创建对象
box = Box.new(10, 20)

# 使用访问器方法
x = box.printWidth()
y = box.printHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

当上面的代码执行时，它会产生以下结果：

```
Width of the box is : 10
Height of the box is : 20
```

与用于访问变量值的访问器方法类似，Ruby 提供了一种在类的外部设置变量值的方式，也就是所谓的设置器方法，定义如下：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 访问器方法
   def getWidth
      @width
   end
   def getHeight
      @height
   end

   # 设置器方法
   def setWidth=(value)
      @width = value
   end
   def setHeight=(value)
      @height = value
   end
end

# 创建对象
box = Box.new(10, 20)

# 使用设置器方法
box.setWidth = 30
box.setHeight = 50

# 使用访问器方法
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

当上面的代码执行时，它会产生以下结果：

```
Width of the box is : 30
Height of the box is : 50
```

实例方法

实例方法的定义与其他方法的定义一样，都是使用 **def** 关键字，但它们只能通过类实例来使用，如下面实例所示。它们的功能不限于访问实例变量，也能按照您的需求做更多其他的任务。

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 创建对象
box = Box.new(10, 20)

# 调用实例方法
a = box.getArea()
puts "Area of the box is : #{a}"
```

当上面的代码执行时，它会产生以下结果：

```
Area of the box is : 200
```

类方法 & 类变量

类变量是在类的所有实例中共享的变量。换句话说，类变量的实例可以被所有的对象实例访问。类变量以两个 @ 字符（@@）作为前缀，类变量必须在类定义中被初始化，如下面实例所示。

类方法使用 **def self.methodname()** 定义，类方法以 end 分隔符结尾。类方法可使用带有类名称的 **classname.methodname** 形式调用，如下面实例所示：

```
#!/usr/bin/ruby -w

class Box
   # 初始化类变量
   @@count = 0
   def initialize(w,h)
      # 给实例变量赋值
      @width, @height = w, h

      @@count += 1
   end

   def self.printCount()
      puts "Box count is : #@@count"
   end
end

# 创建两个对象
box1 = Box.new(10, 20)
box2 = Box.new(30, 100)

# 调用类方法来输出盒子计数
Box.printCount()
```

当上面的代码执行时，它会产生以下结果：

```
Box count is : 2
```

to_s 方法

您定义的任何类都有一个 **to_s** 实例方法来返回对象的字符串表示形式。下面是一个简单的实例，根据 width 和 height 表示 Box 对象：

```
#!/usr/bin/ruby -w

class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 定义 to_s 方法
   def to_s
      "(w:#@width,h:#@height)"  # 对象的字符串格式
   end
end

# 创建对象
box = Box.new(10, 20)

# 自动调用 to_s 方法
puts "String representation of box is : #{box}"
```

当上面的代码执行时，它会产生以下结果：

```
String representation of box is : (w:10,h:20)
```

访问控制

Ruby 为您提供了三个级别的实例方法保护，分别是 **public、private 或 protected**。Ruby 不在实例和类变量上应用任何访问控制。

- **Public 方法：** Public 方法可被任意对象调用。默认情况下，方法都是 public 的，除了 initialize 方法总是 private 的。
- **Private 方法：** Private 方法不能从类外部访问或查看。只有类方法可以访问私有成员。
- **Protected 方法：** Protected 方法只能被类及其子类的对象调用。访问也只能在类及其子类内部进行。

下面是一个简单的实例，演示了这三种修饰符的语法：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 实例方法默认是 public 的
   def getArea
      getWidth() * getHeight
   end

   # 定义 private 的访问器方法
   def getWidth
      @width
   end
   def getHeight
      @height
   end
   # make them private
   private :getWidth, :getHeight

   # 用于输出面积的实例方法
   def printArea
      @area = getWidth() * getHeight
      puts "Big box area is : #@area"
   end
   # 让实例方法是 protected 的
   protected :printArea
end

# 创建对象
box = Box.new(10, 20)

# 调用实例方法
a = box.getArea()
puts "Area of the box is : #{a}"

# 尝试调用 protected 的实例方法
box.printArea()
```

当上面的代码执行时，它会产生以下结果。在这里，第一种方法调用成功，但是第二方法会产生一个问题。

```
Area of the box is : 200
test.rb:42: protected method `printArea' called for #
<Box:0xb7f11280 @height=20, @width=10> (NoMethodError)
```

类的继承

继承，是面向对象编程中最重要的概念之一。继承允许我们根据另一个类定义一个类，这样使得创建和维护应用程序变得更加容易。

继承有助于重用代码和快速执行，不幸的是，Ruby 不支持多继承，但是 Ruby 支持 **mixins**。mixin 就像是多继承的一个特定实现，在多继承中，只有接口部分是可继承的。

当创建类时，程序员可以直接指定新类继承自某个已有类的成员，这样就不用从头编写新的数据成员和成员函数。这个已有类被称为基类或父类，新类被称为派生类或子类。

Ruby 也提供了子类化的概念，子类化即继承，下面的实例解释了这个概念。扩展一个类的语法非常简单。只要添加一个 < 字符和父类的名称到类语句中即可。例如，下面定义了类 *BigBox* 是 *Box* 的子类：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 定义子类
class BigBox < Box

   # 添加一个新的实例方法
   def printArea
      @area = @width * @height
      puts "Big box area is : #@area"
   end
end

# 创建对象
box = BigBox.new(10, 20)

# 输出面积
box.printArea()
```

当上面的代码执行时，它会产生以下结果：

```
Big box area is : 200
```

方法重载

虽然您可以在派生类中添加新的功能，但有时您可能想要改变已经在父类中定义的方法的行为。这时您可以保持方法名称不变，重载方法的功能即可，如下面实例所示：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 定义子类
class BigBox < Box

   # 改变已有的 getArea 方法
   def getArea
      @area = @width * @height
      puts "Big box area is : #@area"
   end
end

# 创建对象
box = BigBox.new(10, 20)

# 使用重载的方法输出面积
box.getArea()
```

运算符重载

我们希望使用 + 运算符执行两个 Box 对象的向量加法，使用 * 运算符来把 Box 的 width 和 height 相乘，使用一元运算符 - 对 Box 的 width 和 height 求反。下面是一个带有数学运算符定义的 Box 类版本：

```
class Box
  def initialize(w,h) # 初始化 width 和 height
    @width,@height = w, h
  end

  def +(other)         # 定义 + 来执行向量加法
    Box.new(@width + other.width, @height + other.height)
  end

  def -@               # 定义一元运算符 - 来对 width 和 height 求反
    Box.new(-@width, -@height)
  end

  def *(scalar)        # 执行标量乘法
    Box.new(@width*scalar, @height*scalar)
  end
end
```

冻结对象

有时候，我们想要防止对象被改变。在 Object 中，freeze 方法可实现这点，它能有效地把一个对象变成一个常量。任何对象都可以通过调用 **Object.freeze** 进行冻结。冻结对象不能被修改，也就是说，您不能改变它的实例变量。

您可以使用 **Object.frozen?** 方法检查一个给定的对象是否已经被冻结。如果对象已被冻结，该方法将返回 true，否则返回一个 false 值。下面的实例解释了这个概念：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 访问器方法
   def getWidth
      @width
   end
   def getHeight
      @height
   end

   # 设置器方法
   def setWidth=(value)
      @width = value
   end
   def setHeight=(value)
      @height = value
   end
end

# 创建对象
box = Box.new(10, 20)

# 让我们冻结该对象
box.freeze
if( box.frozen? )
   puts "Box object is frozen object"
else
   puts "Box object is normal object"
end

# 现在尝试使用设置器方法
box.setWidth = 30
box.setHeight = 50

# 使用访问器方法
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

当上面的代码执行时，它会产生以下结果：

```
Box object is frozen object
test.rb:20:in `setWidth=': can't modify frozen object (TypeError)
        from test.rb:39
```

类常量

您可以在类的内部定义一个常量，通过把一个直接的数值或字符串值赋给一个变量来定义的，常量的定义不需要使用 @ 或 @@。按照惯例，常量的名称使用大写。

一旦常量被定义，您就不能改变它的值，您可以在类的内部直接访问常量，就像是访问变量一样，但是如果您想要在类的外部访问常量，那么您必须使用 **classname::constant**，如下面实例所示。

```
#!/usr/bin/ruby -w

# 定义类
class Box
   BOX_COMPANY = "TATA Inc"
   BOXWEIGHT = 10
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 创建对象
box = Box.new(10, 20)

# 调用实例方法
a = box.getArea()
puts "Area of the box is : #{a}"
puts Box::BOX_COMPANY
puts "Box weight is: #{Box::BOXWEIGHT}"
```

当上面的代码执行时，它会产生以下结果：

```
Area of the box is : 200
TATA Inc
Box weight is: 10
```

类常量可被继承，也可像实例方法一样被重载。

使用 allocate 创建对象

可能有一种情况，您想要在不调用对象构造器 **initialize** 的情况下创建对象，即，使用 new 方法创建对象，在这种情况下，您可以调用 allocate 来创建一个未初始化的对象，如下面实例所示：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   attr_accessor :width, :height

   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 实例方法
   def getArea
      @width * @height
   end
end

# 使用 new 创建对象
box1 = Box.new(10, 20)

# 使用 allocate 创建两一个对象
box2 = Box.allocate

# 使用 box1 调用实例方法
a = box1.getArea()
puts "Area of the box is : #{a}"

# 使用 box2 调用实例方法
a = box2.getArea()
puts "Area of the box is : #{a}"
```

当上面的代码执行时，它会产生以下结果：

```
Area of the box is : 200
test.rb:14: warning: instance variable @width not initialized
test.rb:14: warning: instance variable @height not initialized
test.rb:14:in `getArea': undefined method `*'
   for nil:NilClass (NoMethodError) from test.rb:29
```

类信息

如果类定义是可执行代码，这意味着，它们可在某个对象的上下文中执行，self 必须引用一些东西。让我们来看看下面的实例：.

```
#!/usr/bin/ruby -w

class Box
   # 输出类信息
   puts "Type of self = #{self.type}"
   puts "Name of self = #{self.name}"
end
```

当上面的代码执行时，它会产生以下结果：

```
Type of self = Class
Name of self = Box
```

这意味着类定义可通过把该类作为当前对象来执行，同时也意味着元类和父类中的该方法在方法定义执行期间是可用的。

Ruby 正则表达式

正则表达式是一种特殊序列的字符，它通过使用有专门语法的模式来匹配或查找其他字符串或字符串集合。

语法

正则表达式从字面上看是一种介于斜杠之间或介于跟在 %r 后的任意分隔符之间的模式，如下所示：

```
/pattern/
/pattern/im    # 可以指定选项
%r!/usr/local! # 一般的分隔的正则表达式
```

实例

```
#!/usr/bin/ruby

line1 = "Cats are smarter than dogs";
line2 = "Dogs also like meat";

if ( line1 =~ /Cats(.*)/ )
  puts "Line1 contains Cats"
end
if ( line2 =~ /Cats(.*)/ )
  puts "Line2 contains  Dogs"
end
```

这将产生以下结果：

```
Line1 contains Cats
```

正则表达式修饰符

正则表达式从字面上看可能包含一个可选的修饰符，用于控制各方面的匹配。修饰符在第二个斜杠字符后指定，如上面实例所示。下标列出了 可能的修饰符：

修饰符	描述
i	当匹配文本时忽略大小写。
o	只执行一次 #{} 插值，正则表达式在第一次时就进行判断。
x	忽略空格，允许在正则表达式中进行注释。
m	匹配多行，把换行字符识别为正常字符。
u,e,s,n	把正则表达式解释为 Unicode（UTF-8）、EUC、SJIS 或 ASCII。如果没有指定修饰符，则认为正则表达式使用的是源编码。

就像字符串通过 %Q 进行分隔一样，Ruby 允许您以 %r 作为正则表达式的开头，后面跟着任意分隔符。这在描述包含大量您不想转义的斜杠字符时非常有用。

```
# 下面匹配单个斜杠字符，不转义
%r|/|

# Flag 字符可通过下面的语法进行匹配
%r[</(.*)>]i
```

正则表达式模式

除了控制字符，**(+ ? . * ^ $ ( ) [ ] { } |)**，其他所有字符都匹配本身。您可以通过在控制字符前放置一个反斜杠来对控制字符进行转义。

下表列出了 Ruby 中可用的正则表达式语法。

模式	描述
^	匹配行的开头。
$	匹配行的结尾。
.	匹配除了换行符以外的任意单字符。使用 m 选项时，它也可以匹配换行符。
[...]	匹配在方括号中的任意单字符。
[^...]	匹配不在方括号中的任意单字符。
re*	匹配前面的子表达式零次或多次。
re+	匹配前面的子表达式一次或多次。
re?	匹配前面的子表达式零次或一次。
re{ n}	匹配前面的子表达式 n 次。
re{ n,}	匹配前面的子表达式 n 次或 n 次以上。
re{ n, m}	匹配前面的子表达式至少 n 次至多 m 次。
a| b	匹配 a 或 b。

(re)	对正则表达式进行分组，并记住匹配文本。
(?imx)	暂时打开正则表达式内的 i、 m 或 x 选项。如果在圆括号中，则只影响圆括号内的部分。
(?-imx)	暂时关闭正则表达式内的 i、 m 或 x 选项。如果在圆括号中，则只影响圆括号内的部分。
(?: re)	对正则表达式进行分组，但不记住匹配文本。
(?imx: re)	暂时打开圆括号内的 i、 m 或 x 选项。
(?-imx: re)	暂时关闭圆括号内的 i、 m 或 x 选项。
(?#...)	注释。
(?= re)	使用模式指定位置。没有范围。
(?! re)	使用模式的否定指定位置。没有范围。
(?> re)	匹配无回溯的独立模式。
\w	匹配单词字符。
\W	匹配非单词字符。
\s	匹配空白字符。等价于 [\t\n\r\f]。
\S	匹配非空白字符。
\d	匹配数字。等价于 [0-9]。
\D	匹配非数字。
\A	匹配字符串的开头。
\Z	匹配字符串的结尾。如果存在换行符，则只匹配到换行符之前。
\z	匹配字符串的结尾。
\G	匹配最后一个匹配完成的点。
\b	当在括号外时匹配单词边界，当在括号内时匹配退格键（0x08）。
\B	匹配非单词边界。
\n, \t, etc.	匹配换行符、回车符、制表符，等等。
\1...\9	匹配第 n 个分组子表达式。
\10	如果已匹配过，则匹配第 n 个分组子表达式。否则指向字符编码的八进制表示。

正则表达式实例

字符

实例	描述
/ruby/	匹配 "ruby"
¥	匹配 Yen 符号。Ruby 1.9 和 Ruby 1.8 支持多个字符。

字符类

实例	描述
/[Rr]uby/	匹配 "Ruby" 或 "ruby"
/rub[ye]/	匹配 "ruby" 或 "rube"
/[aeiou]/	匹配任何一个小写元音字母
/[0-9]/	匹配任何一个数字，与 /[0123456789]/ 相同
/[a-z]/	匹配任何一个小写 ASCII 字母
/[A-Z]/	匹配任何一个大写 ASCII 字母
/[a-zA-Z0-9]/	匹配任何一个括号内的字符
/[^aeiou]/	匹配任何一个非小写元音字母的字符
/[^0-9]/	匹配任何一个非数字字符

特殊字符类

实例	描述
/./	匹配除了换行符以外的其他任意字符
/./m	在多行模式下，也能匹配换行符
/\d/	匹配一个数字，等同于 /[0-9]/
/\D/	匹配一个非数字，等同于 /[^0-9]/
/\s/	匹配一个空白字符，等同于 /[\t\r\n\f]/
/\S/	匹配一个非空白字符，等同于 /[^ \t\r\n\f]/
/\w/	匹配一个单词字符，等同于 /[A-Za-z0-9_]/
/\W/	匹配一个非单词字符，等同于 /[^A-Za-z0-9_]/

重复

实例	描述
/ruby?/	匹配 "rub" 或 "ruby"。其中，y 是可有可无的。
/ruby*/	匹配 "rub" 加上 0 个或多个的 y。
/ruby+/	匹配 "rub" 加上 1 个或多个的 y。
/\d{3}/	刚好匹配 3 个数字。
/\d{3,}/	匹配 3 个或多个数字。
/\d{3,5}/	匹配 3 个、4 个或 5 个数字。

非贪婪重复

这会匹配最小次数的重复。

实例	描述
/<.*>/	贪婪重复：匹配 " <ruby>perl> "
/<.*?>/	非贪婪重复：匹配 " <ruby>perl> " 中的 " <ruby> "

通过圆括号进行分组

实例	描述
/\D\d+/	无分组： + 重复 \d
/(\D\d)+/	分组： + 重复 \D\d 对
/([Rr]uby(,)?)+/	匹配 "Ruby"、"Ruby, ruby, ruby"，等等

反向引用

这会再次匹配之前匹配过的分组。

实例	描述
/([Rr])uby&\1ails/	匹配 ruby&rails 或 Ruby&Rails
/(['"])(?:(?!\1).)*\1/	单引号或双引号字符串。\1 匹配第一个分组所匹配的字符，\2 匹配第二个分组所匹配的字符，依此类推。

替换

实例	描述
/ruby|rube/	匹配 "ruby" 或 "rube"
/rub(y|le))/	匹配 "ruby" 或 "ruble"
/ruby(!+|\?)/	"ruby" 后跟一个或多个 ! 或者跟一个 ?

锚

这需要指定匹配位置。

实例	描述
/^Ruby/	匹配以 "Ruby" 开头的字符串或行
/Ruby$/	匹配以 "Ruby" 结尾的字符串或行
/\ARuby/	匹配以 "Ruby" 开头的字符串
/Ruby\Z/	匹配以 "Ruby" 结尾的字符串
/\bRuby\b/	匹配单词边界的 "Ruby"
/\brub\B/	\B 是非单词边界：匹配 "rube" 和 "ruby" 中的 "rub"，但不匹配单独的 "rub"
/Ruby(?=!)/	如果 "Ruby" 后跟着一个感叹号，则匹配 "Ruby"
/Ruby(?!!)/	如果 "Ruby" 后没有跟着一个感叹号，则匹配 "Ruby"

圆括号的特殊语法

实例	描述
/R(?#comment)/	匹配 "R"。所有剩余的字符都是注释。
/R(?i)uby/	当匹配 "uby" 时不区分大小写。
/R(?i:uby)/	与上面相同。
/rub(?:y|le))/	只分组，不进行 \1 反向引用

搜索和替换

sub 和 **gsub** 及它们的替代变量 **sub!** 和 **gsub!** 是使用正则表达式时重要的字符串方法。

所有这些方法都是使用正则表达式模式执行搜索与替换操作。**sub** 和 **sub!** 替换模式的第一次出现，**gsub** 和 **gsub!** 替换模式的所有出现。

sub 和 **gsub** 返回一个新的字符串，保持原始的字符串不被修改，而 **sub!** 和 **gsub!** 则会修改它们调用的字符串。

下面是一个实例：

```
#!/usr/bin/ruby

phone = "2004-959-559 #This is Phone Number"

# 删除 Ruby 的注释
phone = phone.sub!(/#.*$/, "")
puts "Phone Num : #{phone}"

# 移除数字以外的其他字符
phone = phone.gsub!(/\D/, "")
puts "Phone Num : #{phone}"
```

这将产生以下结果：

```
Phone Num : 2004-959-559
Phone Num : 2004959559
```

下面是另一个实例：

```
#!/usr/bin/ruby

text = "rails are rails, really good Ruby on Rails"

# 把所有的 "rails" 改为 "Rails"
text.gsub!("rails", "Rails")

# 把所有的单词 "Rails" 都改成首字母大写
text.gsub!(/\brails\b/, "Rails")

puts "#{text}"
```

这将产生以下结果：

```
Rails are Rails, really good Ruby on Rails
```

Ruby 数据库访问 - DBI 教程

本章节将向您讲解如何使用 Ruby 访问数据库。*Ruby DBI* 模块为 Ruby 脚本提供了类似于 Perl DBI 模块的独立于数据库的接口。

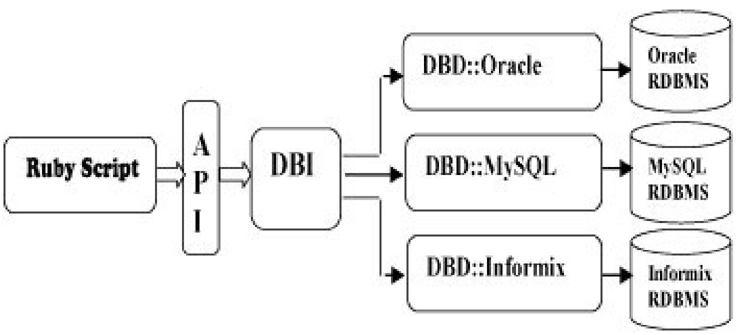
DBI 即 Database independent interface，代表了 Ruby 独立于数据库的接口。DBI 在 Ruby 代码与底层数据库之间提供了一个抽象层，允许您简单地实现数据库切换。它定义了一系列方法、变量和规范，提供了一个独立于数据库的一致的数据库接口。

DBI 可与下列进行交互：

- ADO (ActiveX Data Objects)
- DB2
- Frontbase
- mSQL
- MySQL
- ODBC
- Oracle
- OCI8 (Oracle)
- PostgreSQL
- Proxy/Server
- SQLite
- SQLRelay

DBI 应用架构

DBI 独立于任何在后台中可用的数据库。无论您使用的是 Oracle、MySQL、Informix，您都可以使用 DBI。下面的架构图清晰地说明了这点。



Ruby DBI 一般的架构使用两个层：

- 数据库接口（DBI）层。该层是独立于数据库，并提供了一系列公共访问方法，方法的使用不分数据库服务器类型。

- 数据库驱动（DBD）层。该层是依赖于数据库，不同的驱动提供了对不同的数据库引擎的访问。MySQL、PostgreSQL、InterBase、Oracle 等分别使用不同的驱动。每个驱动都负责解释来自 DBI 层的请求，并把这些请求映射为适用于给定类型的数据库服务器的请求。

先决条件

如果您想要编写 Ruby 脚本来访问 MySQL 数据库，您需要先安装 Ruby MySQL 模块。

该模块是一个 DBD，可从 http://www.tmtm.org/en/mysql/ruby/ 上下载。

获取并安装 Ruby/DBI

您可以从下面的链接下载并安装 Ruby DBI 模块：

http://rubyforge.org/projects/ruby-dbi/

在开始安装之前，请确保您拥有 root 权限。现在，请安装下面的步骤进行安装：

步骤 1

```
$ tar zxf dbi-0.2.0.tar.gz
```

步骤 2

进入目录 *dbi-0.2.0*，在目录中使用 *setup.rb* 脚本进行配置。最常用的配置命令是 config 参数后不跟任何参数。该命令默认配置为安装所有的驱动。

```
$ ruby setup.rb config
```

更具体地，您可以使用 --with 选项来列出了您要使用的特定部分。例如，如果只想配置主要的 DBI 模块和 MySQL DBD 层驱动，请输入下面的命令：

```
$ ruby setup.rb config --with=dbi,dbd_mysql
```

步骤 3

最后一步是建立驱动器，使用下面命令进行安装：

```
$ ruby setup.rb setup
$ ruby setup.rb install
```

数据库连接

假设我们使用的是 MySQL 数据库，在连接数据库之前，请确保：

- 您已经创建了一个数据库 TESTDB。
- 您已经在 TESTDB 中创建了表 EMPLOYEE。
- 该表带有字段 FIRST_NAME、LAST_NAME、AGE、SEX 和 INCOME。
- 设置用户 ID "testuser" 和密码 "test123" 来访问 TESTDB
- 已经在您的机器上正确地安装了 Ruby 模块 DBI。
- 您已经看过 MySQL 教程，理解了 MySQL 基础操作。

下面是连接 MySQL 数据库 "TESTDB" 的实例：

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     # 获取服务器版本字符串，并显示
     row = dbh.select_one("SELECT VERSION()")
     puts "Server version: " + row[0]
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

当运行这段脚本时，将会在 Linux 机器上产生以下结果。

```
Server version: 5.0.45
```

如果建立连接时带有数据源，则返回数据库句柄（Database Handle），并保存到 **dbh** 中以便后续使用，否则 **|dbh** 将被设置为 nil 值，*e.err* 和 *e::errstr* 分别返回错误代码和错误字符串。

最后，在退出这段程序之前，请确保关闭数据库连接，释放资源。

INSERT 操作

当您想要在数据库表中创建记录时，需要用到 INSERT 操作。

一旦建立了数据库连接，我们就可以准备使用 **do** 方法或 **prepare** 和 **execute** 方法创建表或创建插入数据表中的记录。

使用 do 语句

不返回行的语句可通过调用 **do** 数据库处理方法。该方法带有一个语句字符串参数，并返回该语句所影响的行数。

```
dbh.do("DROP TABLE IF EXISTS EMPLOYEE")
dbh.do("CREATE TABLE EMPLOYEE (
     FIRST_NAME  CHAR(20) NOT NULL,
     LAST_NAME  CHAR(20),
     AGE INT,
     SEX CHAR(1),
     INCOME FLOAT )" );
```

同样地，您可以执行 SQL *INSERT* 语句来创建记录插入 EMPLOYEE 表中。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     dbh.do( "INSERT INTO EMPLOYEE(FIRST_NAME,
                   LAST_NAME,
                   AGE,
           SEX,
           INCOME)
          VALUES ('Mac', 'Mohan', 20, 'M', 2000)" )
     puts "Record has been created"
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

使用 *prepare* 和 *execute*

您可以使用 DBI 的 *prepare* 和 *execute* 方法来执行 Ruby 代码中的 SQL 语句。

创建记录的步骤如下：

- 准备带有 INSERT 语句的 SQL 语句。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** API 来完成。

- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

下面是使用这两种方法的语法：

```
sth = dbh.prepare(statement)
sth.execute
   ... zero or more SQL operations ...
sth.finish
```

这两种方法可用于传 **bind** 值给 SQL 语句。有时候被输入的值可能未事先给出，在这种情况下，则会用到绑定值。使用问号（**?**）替代实际值，实际值通过 execute() API 来传递。

下面的实例在 EMPLOYEE 表中创建了两个记录：

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare( "INSERT INTO EMPLOYEE(FIRST_NAME,
                   LAST_NAME,
                   AGE,
            SEX,
           INCOME)
                   VALUES (?, ?, ?, ?, ?)" )
     sth.execute('John', 'Poul', 25, 'M', 2300)
     sth.execute('Zara', 'Ali', 17, 'F', 1000)
     sth.finish
     dbh.commit
     puts "Record has been created"
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

如果同时使用多个 INSERT，那么先准备一个语句，然后在一个循环中多次执行它要比通过循环每次调用 do 有效率得多。

READ 操作

对任何数据库的 READ 操作是指从数据库中获取有用的信息。

一旦建立了数据库连接，我们就可以准备查询数据库。我们可以使用 **do** 方法或 **prepare** 和 **execute** 方法从数据库表中获取值。

获取记录的步骤如下：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。
- 逐一获取结果，并输出这些结果。这将通过使用 **fetch** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** 方法来完成。

下面的实例从 EMPLOYEE 表中查询所有工资（salary）超过 1000 的记录。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                        "testuser", "test123")
     sth = dbh.prepare("SELECT * FROM EMPLOYEE
                        WHERE INCOME > ?")
     sth.execute(1000)

     sth.fetch do |row|
        printf "First Name: %s, Last Name : %s\n", row[0], row[1]
        printf "Age: %d, Sex : %s\n", row[2], row[3]
        printf "Salary :%d \n\n", row[4]
     end
     sth.finish
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

这将产生以下结果：

```
First Name: Mac, Last Name : Mohan
Age: 20, Sex : M
Salary :2000

First Name: John, Last Name : Poul
Age: 25, Sex : M
Salary :2300
```

还有很多从数据库获取记录的方法，如果您感兴趣，可以查看 Ruby DBI Read 操作。

Update 操作

对任何数据库的 UPDATE 操作是指更新数据库中一个或多个已有的记录。下面的实例更新 SEX 为 'M' 的所有记录。在这里，我们将把所有男性的 AGE 增加一岁。这将分为三步：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。

- 释放语句句柄。这将通过使用 **finish** 方法来完成。
- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("UPDATE EMPLOYEE SET AGE = AGE + 1
                       WHERE SEX = ?")
     sth.execute('M')
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

DELETE 操作

当您想要从数据库中删除记录时，需要用到 DELETE 操作。下面的实例从 EMPLOYEE 中删除 AGE 超过 20 的所有记录。该操作的步骤如下：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中删除所需的记录。这将通过使用 **execute** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** 方法来完成。
- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("DELETE FROM EMPLOYEE
                       WHERE AGE > ?")
     sth.execute(20)
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

执行事务

事务是一种确保交易一致性的机制。事务应具有下列四种属性：

- 原子性（**Atomicity**）：事务的原子性指的是，事务中包含的程序作为数据库的逻辑工作单位，它所做的对数据修改操作要么全部执行，要么完全不执行。
- 一致性（**Consistency**）：事务的一致性指的是在一个事务执行之前和执行之后数据库都必须处于一致性状态。假如数据库的状态满足所有的完整性约束，就说该数据库是一致的。
- 隔离性（**Isolation**）：事务的隔离性指并发的事务是相互隔离的，即一个事务内部的操作及正在操作的数据必须封锁起来，不被其它企图进行修改的事务看到。
- 持久性（**Durability**）：事务的持久性意味着当系统或介质发生故障时，确保已提交事务的更新不能丢失。即一旦一个事务提交，它对数据库中数据的改变应该是永久性的，耐得住任何数据库系统故障。持久性通过数据库备份和恢复来保证。

DBI 提供了两种执行事务的方法。一种是 *commit* 或 *rollback* 方法，用于提交或回滚事务。还有一种是 *transaction* 方法，可用于实现事务。接下来我们来介绍这两种简单的实现事务的方法：

方法 I

第一种方法使用 DBI 的 *commit* 和 *rollback* 方法来显式地提交或取消事务：

```
dbh['AutoCommit'] = false # 设置自动提交为 false.
begin
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'Zara'")
  dbh.commit
rescue
  puts "transaction failed"
  dbh.rollback
end
dbh['AutoCommit'] = true
```

方法 II

第二种方法使用 *transaction* 方法。这个方法相对简单些，因为它需要一个包含构成事务语句的代码块。*transaction* 方法执行块，然后根据块是否执行成功，自动调用 *commit* 或 *rollback*：

```
dbh['AutoCommit'] = false # 设置自动提交为 false
dbh.transaction do |dbh|
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'Zara'")
end
dbh['AutoCommit'] = true
```

COMMIT 操作

Commit 是一种标识数据库已完成更改的操作，在这个操作后，所有的更改都不可恢复。

下面是一个调用 **commit** 方法的简单实例。

```
dbh.commit
```

ROLLBACK 操作

如果您不满意某个或某几个更改，您想要完全恢复这些更改，则使用 **rollback** 方法。

下面是一个调用 **rollback** 方法的简单实例。

```
dbh.rollback
```

断开数据库

如需断开数据库连接，请使用 disconnect API。

```
dbh.disconnect
```

如果用户通过 disconnect 方法关闭了数据库连接，DBI 会回滚所有未完成的事务。但是，不需要依赖于任何 DBI 的实现细节，您的应用程序就能很好地显式调用 commit 或 rollback。

处理错误

有许多不同的错误来源。比如在执行 SQL 语句时的语法错误，或者是连接失败，又或者是对一个已经取消的或完成的语句句柄调用 fetch 方法。

如果某个 DBI 方法失败，DBI 会抛出异常。DBI 方法会抛出任何类型的异常，但是最重要的两种异常类是 *DBI::InterfaceError* 和 *DBI::DatabaseError*。

这些类的 Exception 对象有 *err*、*errstr* 和 *state* 三种属性，分表代表了错误号、一个描述性的错误字符串和一个标准的错误代码。属性具体说明如下：

- **err**：返回所发生的错误的整数表示法，如果 DBD 不支持则返回 *nil*。例如，Oracle DBD 返回 *ORA-XXXX* 错误消息的数字部分。
- **errstr**：返回所发生的错误的字符串表示法。
- **state**：返回所发生的错误的 SQLSTATE 代码。SQLSTATE 是五字符长度的字符串。大多数的 DBD 并不支持它，所以会返回 nil。

在上面的实例中您已经看过下面的代码：

```
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

为了获取脚本执行时有关脚本执行内容的调试信息，您可以启用跟踪。为此，您必须首先下载 dbi/trace 模块，然后调用控制跟踪模式和输出目的地的 *trace* 方法：

```
require "dbi/trace"
..............

trace(mode, destination)
```

mode 的值可以是 0（off）、1、2 或 3，destination 的值应该是一个 IO 对象。默认值分别是 2 和 STDERR。

方法的代码块

有一些创建句柄的方法。这些方法通过代码块调用。使用带有方法的代码块的优点是，它们为代码块提供了句柄作为参数，当块终止时会自动清除句柄。下面是一些实例，有助于理解这个概念。

- **DBI.connect** ：该方法生成一个数据库句柄，建议在块的末尾调用 *disconnect* 来断开数据库。
- **dbh.prepare** ：|该方法生成一个语句句柄，建议在块的末尾调用 *finish*。在块内，您必须调用 *execute* 方法来执行语句。
- **dbh.execute** ：|该方法与 dbh.prepare 类似，但是 dbh.execute 不需要在块内调用 execute 方法。语句句柄会自动执行。

实例 1

DBI.connect 可带有一个代码块，向它传递数据库句柄，且会在块的末尾自动断开句柄。

```
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                  "testuser", "test123") do |dbh|
```

实例 2

dbh.prepare | 可带有一个代码块，向它传递语句句柄，且会在块的末尾自动调用 finish。

```
dbh.prepare("SHOW DATABASES") do |sth|
       sth.execute
       puts "Databases: " + sth.fetch_all.join(", ")
end
```

实例 3

dbh.execute | 可带有一个代码块，向它传递语句句柄，且会在块的末尾自动调用 finish。

```
dbh.execute("SHOW DATABASES") do |sth|
   puts "Databases: " + sth.fetch_all.join(", ")
end
```

DBI *transaction* 方法也可带有一个代码块，这在上面的章节中已经讲解过了。

特定驱动程序的函数和属性

DBI 让数据库驱动程序提供了额外的特定数据库的函数，这些函数可被用户通过任何 Handle 对象的 *func* 方法进行调用。

使用 **[]=** or **[]** 方法可以设置或获取特定驱动程序的属性。

DBD::Mysql 实现了下列特定驱动程序的函数：

函数	描述
dbh.func(:createdb, db_name)	创建一个新的数据库。
dbh.func(:dropdb, db_name)	删除一个数据库。
dbh.func(:reload)	执行重新加载操作。
dbh.func(:shutdown)	关闭服务器。
dbh.func(:insert_id) => Fixnum	返回该连接的最近 AUTO_INCREMENT 值。
dbh.func(:client_info) => String	根据版本返回 MySQL 客户端信息。
dbh.func(:client_version) => Fixnum	根据版本返回客户端信息。这与 :client_info 类似，但是它会返回一个 fixnum，而不是返回字符串。
dbh.func(:host_info) => String	返回主机信息。
dbh.func(:proto_info) => Fixnum	返回用于通信的协议。
dbh.func(:server_info) => String	根据版本返回 MySQL 服务器端信息。
dbh.func(:stat) => Stringb> 返回数据库的当前状态。	
dbh.func(:thread_id) => Fixnum	返回当前线程的 ID。

实例

```
#!/usr/bin/ruby

require "dbi"
begin
   # 连接到 MySQL 服务器
   dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                     "testuser", "test123")
   puts dbh.func(:client_info)
   puts dbh.func(:client_version)
   puts dbh.func(:host_info)
   puts dbh.func(:proto_info)
   puts dbh.func(:server_info)
   puts dbh.func(:thread_id)
   puts dbh.func(:stat)
rescue DBI::DatabaseError => e
   puts "An error occurred"
   puts "Error code:    #{e.err}"
   puts "Error message: #{e.errstr}"
ensure
   dbh.disconnect if dbh
end
```

这将产生以下结果：

```
5.0.45
50045
Localhost via UNIX socket
10
5.0.45
150621
Uptime: 384981  Threads: 1  Questions: 1101078  Slow queries: 4 \
Opens: 324  Flush tables: 1  Open tables: 64  \
Queries per second avg: 2.860
```

Ruby CGI 编程

Ruby 是一门通用的语言，不仅仅是一门应用于WEB开发的语言，但 Ruby 在WEB应用及WEB工具中的开发是最常见的。

使用Ruby您不仅可以编写自己的SMTP服务器，FTP程序，或Ruby Web服务器，而且还可以使用Ruby进行CGI编程。

接下来，让我们花点时间来学校Ruby的CGI编辑。

编写 CGI 脚本

最脚本的 Ruby CGI 代码如下所示：

```
#!/usr/bin/ruby puts "HTTP/1.0 200 OK" puts "Content-type: text/html\n\n" puts "This is a
```

你可以将该代码保持到 test.cgi 文件中，上次到服务器并赋予足够权限，即可作为 CGI 脚本执行。

如果你站的的地址为http://www.example.com/ ，即可用过http://www.example.com/test.cgi 访问该程序，输出结果为： "This is a test."。

浏览器访问该网址后，Web 服务器会在站点目录下找到 test.cgi文件，然后通过Ruby解析器来解析脚本代码并访问HTML文档。

使用 cgi.rb

Ruby 可以调用 CGI 库来编写更复杂的CGI脚本。

以下代码调用了 CGI 库来创建一个脚本的CGI脚本。

```
#!/usr/bin/ruby

require 'cgi'

cgi = CGI.new
puts cgi.header
puts "<html><body>This is a test</body></html>"
```

以下代码中，创建了CGI 对象并打印头部信息。

表单处理

使用CGI库可以通过两种方式获取表单提交（或URL中的参数）的数据， 例如URL：/cgi-bin/test.cgi?FirstName=Zara&LastName=Ali。

你可以使用 CGI#[] 来直接获取参数FirstName和LastName：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi['FirstName'] # =>  ["Zara"]
cgi['LastName']  # =>  ["Ali"]
```

另外一种获取表单数据的方法：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
h = cgi.params  # =>  {"FirstName"=>["Zara"],"LastName"=>["Ali"]}
h['FirstName']  # =>  ["Zara"]
h['LastName']   # =>  ["Ali"]
```

以下代码用于检索所有的键值：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi.keys         # =>  ["FirstName", "LastName"]
```

如果表单包含了多个相同名称的字段，则该相同字段的值将保存在数组中。

以下实例中，指定表单中三个相同的字段"name"， 值分别为 "Zara", "Huma" 和 "Nuha":

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi['name']        # => "Zara"
cgi.params['name'] # => ["Zara", "Huma", "Nuha"]
cgi.keys           # => ["name"]
cgi.params         # => {"name"=>["Zara", "Huma", "Nuha"]}
```

注意：Ruby 会自动判断 GET 和 POST 方法，所以无需对两种方法区别对待。

以下是相关的HML代码：

```
<html>
<body>
<form method="POST" action="http://www.example.com/test.cgi">
First Name :<input type="text" name="FirstName" value="" />
<br />
Last Name :<input type="text" name="LastName" value="" />

<input type="submit" value="Submit Data" />
</form>
</body>
</html>
```

创建 Form 表单和 HTML

CGI 包含了大量的方法来创建 HTML，每个HTML标签都有相对应的方法。 在使用这些方法前，比必须通过 CGI.new 来创建 CGI 对象。

为了使标签的嵌套更加的简单，这些方法将内容作为了代码块，代码块将返回字符串作为标签的内容。如下所示：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cgi.out{
   cgi.html{
      cgi.head{ "\n"+cgi.title{"This Is a Test"} } +
      cgi.body{ "\n"+
         cgi.form{"\n"+
            cgi.hr +
            cgi.h1 { "A Form: " } + "\n"+
            cgi.textarea("get_text") +"\n"+
            cgi.br +
            cgi.submit
         }
      }
   }
}
```

字符串转义

当你在处理 URL 中的参数或者 HTML 表单数据时，需要对指定的特殊字符进行转义，如：引号（"）,反斜杠(/)。

Ruby CGI 对象提供了CGI.escape 和 CGI.unescape 方法来处理这些特殊字符的转义：

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escape(Zara Ali/A Sweet & Sour Girl")
```

以上代码执行结果如下：

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escape(Zara Ali/A Sweet & Sour Girl")
```

另一组实例：

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escapeHTML('<h1>Zara Ali/A Sweet & Sour Girl</h1>')
```

以上代码执行结果如下：

```
&lt;h1&gt;Zara Ali/A Sweet & Sour Girl&lt;/h1&gt;'
```

CGI 类中常用的方法

以下是Ruby中完整的CGI类的相关方法

- Ruby CGI - 标准 CGI 库相关方法

Cookies 和 Sessions

- Ruby CGI Cookies - 如何处理 CGI Cookies.
- Ruby CGI Sessions - 如何处理 CGI sessions.

Ruby CGI方法

以下为CGI类的方法列表：

序号	方法描述
CGI::new([level="query"])	创建 CGI 对象。query可以是以下值: **query:** 没有 HTML 生成输出 **html3:** HTML3.2 **html4:** HTML4.0 Strict **html4Tr:** HTML4.0 Transitional **html4Fr:** HTML4.0 Frameset
CGI::escape(str)	使用 URL 编码来转义字符串
CGI::unescape(str)	对通过 escape() 编码的字符串进行解码。
CGI::escapeHTML(str)	编码 HTML 特殊字符, 包括: & < >。
CGI::unescapeHTML(str)	解码 HTML 特殊字符, 包括: & < >。
CGI::escapeElement(str[, element...])	在指定的 HTML 元素中编码 HTML 特殊字符。
CGI::unescapeElement(str, element[, element...])	在指定的 HTML 元素中解码 HTML 特殊字符。
CGI::parse(query)	解析查询字符串，并返回包含哈希的 键=》值 对。
CGI::pretty(string[, leader=" "])	返回整齐的HTML格式。 如果指定了 *leader*， 它将写入到每一行的开头。 *leader* 默认值为两个空格。
CGI::rfc1123_date(time)	根据 RFC-1123 来格式化时间 (例如, Tue, 2 Jun 2008 00:00:00 GMT)。

CGI 实例化方法

以下实例中我们将 CGI::new 的对象赋值给 c 变量，方法列表如下：

序号	方法描述
c[name]	返回一个数组，包含了对应字段名为 *name* 的值。
c.checkbox(name[, value[, check=false]]) c.checkbox(options)	返回 HTML 字符串用于定义 checkbox 字段。标签的属性可以以一个哈希函数作为参数传递。
c.checkbox_group(name, value...) c.checkbox_group(options)	>返回 HTML 字符串用于定义 checkbox 组。标签的属性可以以一个哈希函数作为参数传递。
c.file_field(name[, size=20[,	

max]]) c.file_field(options)	
c.form([method="post"[, url]]) { ...} c.form(options)	返回定义 form 表单的HTML字符串。 如果指定了代码块，将作为表单内容输出。标签的属性可以以一个哈希函数作为参数传递。
c.cookies	返回 CGI::Cookie 对象，包含了cookie 中的键值对。
c.header([header])	返回 CGI 头部的信息。如果 header 参数是哈希值，其键 - 值对，用于创建头部信息。
c.hidden(name[, value]) c.hidden(options)	返回定义一个隐藏字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.image_button(url[, name[, alt]]) c.image_button(options)	返回定义一个图像按钮的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.keys	返回一个数组，包含了表单的字段名。
c.key?(name) c.has_key?(name) c.include?(name)	如果表单包含了指定的字段名返回 true。
c.multipart_form([url[, encode]]) { ...} c.multipart_form(options) { ...}	返回定义一个多媒体表单（multipart）的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.out([header]) { ...}	生成 HTML 并输出。使用由块的输出来创建页面的主体生成的字符串。
c.params	返回包含表单字段名称和值的哈希值。
c.params= hash	设置使用字段名和值。
c.password_field(name[, value[, size=40[, max]]]) c.password_field(options)	返回定义一个password字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.popup_menu(name, value...) c.popup_menu(options) c.scrolling_list(name, value...) c.scrolling_list(options)	返回定义一个弹出式菜单的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.radio_button(name[, value[, checked=false]]) c.radio_button(options)	返回定义一个radio字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.radio_group(name, value...) c.radio_group(options)	返回定义一个radio按钮组的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.reset(name[, value]) c.reset(options)	返回定义一个reset按钮的HTML字符串。 标签的属性可以以一个哈希函数作为参数传递
c.text_field(name[, value[, size=40[, max]]]) c.text_field(options)	返回定义一个text字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.textarea(name[, cols=70[,	返回定义一个textarea字段的HTML字符串。 如果指定了块，代码块输出的字符串将作为

rows=10]]) { ...} c.textarea(options) { ...}	果指定了块，代码块输出的字符串将作为 textarea 的内容。 标签的属性可以以一个哈希函数作为参数传递。

HTML 生成方法

你可以再 CGI 实例中使用相应的 HTML 标签名来创建 HTML 标签，实例如下：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cgi.out{
   cgi.html{
      cgi.head{ "\n"+cgi.title{"This Is a Test"} } +
      cgi.body{ "\n"+
         cgi.form{"\n"+
            cgi.hr +
            cgi.h1 { "A Form: " } + "\n"+
            cgi.textarea("get_text") +"\n"+
            cgi.br +
            cgi.submit
         }
      }
   }
}
```

CGI 对象属性

你可以再 CGI 实例中使用以下属性：

属性	返回值
accept	可接受的 MIME 类型
accept_charset	可接受的字符集
accept_encoding	可接受的编码
accept_language	可接受的语言
auth_type	可接受的类型
raw_cookie	Cookie 数据 (原字符串)
content_length	内容长度（Content length）
content_type	内容类型（Content type）
From	Client e-mail 地址
gateway_interface	CGI 版本
path_info	路径
path_translated	转换后的路径
Query_string	查询字符串
referer	之前访问网址
remote_addr	客户端主机地址(IP)
remote_host	客户端主机名
remote_ident	客户端名
remote_user	经过身份验证的用户
request_method	请求方法(GET, POST, 等。)
script_name	参数名
server_name	服务器名
server_port	服务器端口
server_protocol	服务器协议
server_software	服务器软件
user_agent	用户代理（User agent）

Ruby CGI Cookies

HTTP协议是无状态协议。但对于一个商业网站，它需要保持不同的页面间的会话信息。

如用户在网站注册过程中需要跳转页面，但又要保证之前填写的信息部丢失。

这种情况下 Cookie 很好的帮我们解决了问题。

Cookie 是如何工作的？

几乎所有的网站设计者在进行网站设计时都使用了Cookie，因为他们都想给浏览网站的用户提供一个更友好的、人文化的浏览环境，同时也能更加准确地收集访问者的信息。

写入和读取

Cookies集合是附属于Response对象及Request对象的数据集合，使用时需要在前面加上Response或Request。

用于给客户机发送Cookies的语法通常为：

当给不存在的Cookies集合设置时，就会在客户机创建，如果该Cookies己存在，则会被代替。由于Cookies是作为HTTP传输的头信息的一部分发给客户机的，所以向客户机发送Cookies的代码一般放在发送给浏览器的HTML文件的标记之前。

如果用户要读取Cookies，则必须使用Request对象的Cookies集合，其使用方法是： 需要注意的是，只有在服务器未被下载任何数据给浏览器前，浏览器才能与Server进行Cookies集合的数据交换，一旦浏览器开始接收Server所下载的数据，Cookies的数据交换则停止，为了避免错误，要在程序和前面加上response.Buffer=True。

集合的属性

- **1.Expires**属性：此属性用来给Cookies设置一个期限，在期限内只要打开网页就可以调用被保存的Cookies，如果过了此期限Cookies就自动被删除。如： 设定Cookies的有效期到2004年4月1日，到时将自动删除。如果一个Cookies没有设定有效期，则其生命周期从打开浏览器开始，到关闭浏览器结束，每次运行后生命周期将结束，下次运行将重新开始。
- **2.Domain**属性：这个属性定义了Cookies传送数据的唯一性。若只将某Cookies传送给_blank">搜狐主页时，则可使用如下代码：
- **3.Path**属性：定义了Cookies只发给指定的路径请求，如果Path属性没有被设置，则使用应用软件的缺省路径。
- **4.Secure**属性：指定Cookies能否被用户读取。

- **5、Name=Value :** Cookies是以键值对的形式进行设置和检索的。

Ruby 中处理Cookies

你可以创建一个名为 cookie 的对象并存储文本信息，将该信息发送至浏览器，调用 CGI.out 设置cookie的头部：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = CGI::Cookie.new('name' => 'mycookie',
                         'value' => 'Zara Ali',
                         'expires' => Time.now + 3600)
cgi.out('cookie' => cookie) do
   cgi.head + cgi.body { "Cookie stored" }
end
```

接下来我们回到这个页面，并查找cookie值，如下所示：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = cgi.cookies['mycookie']
cgi.out('cookie' => cookie) do
   cgi.head + cgi.body { cookie[0] }
end
```

CGI::Cookie对象实例化时包含以下参数：

参数	描述
name	规定 cookie 的名称。
value	规定 cookie 的值。
expire	规定 cookie 的有效期。
path	规定 cookie 的服务器路径。
domain	规定 cookie 的域名。
secure	规定是否通过安全的 HTTPS 连接来传输 cookie。

Ruby CGI Sessions

CGI::Session 可以为用户和CGI环境保存持久的会话状态，会话使用后需要关闭，这样可以保证数据写入到存储当中，当会话完成后，你需要删除该数据。

```
#!/usr/bin/ruby

require 'cgi'
require 'cgi/session'
cgi = CGI.new("html4")

sess = CGI::Session.new( cgi, "session_key" => "a_test",
                              "prefix" => "rubysess.")
lastaccess = sess["lastaccess"].to_s
sess["lastaccess"] = Time.now
if cgi['bgcolor'][0] =~ /[a-z]/
  sess["bgcolor"] = cgi['bgcolor']
end

cgi.out{
  cgi.html {
    cgi.body ("bgcolor" => sess["bgcolor"]){
      "The background of this page"    +
      "changes based on the 'bgcolor'" +
      "each user has in session."      +
      "Last access time: #{lastaccess}"
    }
  }
}
```

访问 "/cgi-bin/test.cgi?bgcolor=red" 将跳转到指定背景颜色的页面。

会话数据存在在服务器的临时文件目录中，prefix 参数指定了会话的前缀，将作为临时文件的前缀。这样你在服务器上可以轻松的识别不同的会话临时文件。

CGI::Session 类

CGI::Session 保持了用户与 CGI 环境的持久状态。 会话可以在内存中，也可以在硬盘上。

类方法

Ruby 类 Class CGI::Session 提供了简单的方法来创建 session:

```
CGI::Session::new( cgi[, option])
```

启用一个新的 CGI 会话并返回相应的 CGI::Session 对象。选项可以是可选的哈希，可以是以下值：

- **session_key:** 键名保存会话 默认为 _session_id。

- **session_id:** 唯一的会话 ID。自动生成
- **new_session:** 如果为true，为当前会话创建一个新的Session id。 如果为 false, 通过 session_id 使用已存在的 session 标识。 如果省略该参数，如果可用则使用现有的会话，否则创建一个新的。
- **database_manager:** 用于保存 sessions 的类，可以是 CGI::Session::FileStore or CGI::Session::MemoryStore。默认为 FileStore。
- **tmpdir:** 对于 FileStore, 为 session 的错存储目录。
- **prefix:** 对于 FileStore, 为 session 文件的前缀。

实例化方法

方法	描述
[ ]	返回给定 key 的值。查看实例。
[ ]=	设置给定 key 的值。 查看实例。
delete	调用底层数据库管理的删除方法。对于 FileStore, 删除包含 session 的物理文件。 对于 MemoryStore, 从内存中移除 session 数据。
update	调用底层数据库管理的更新方法。 对于 FileStore, 将 session 写入到磁盘中。 对于 MemoryStore则无效果。

Ruby 发送邮件 - SMATP

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

Ruby提供了 Net::SMTP 来发送邮件，并提供了两个方法 new 和 start:

- **new** 方法有两个参数：
  - *server name* 默认为 localhost
  - *port number* 默认为 25
- **start** 方法有以下参数：
  - *server* - SMTP 服务器 IP, 默认为 localhost
  - *port* - 端口号，默认为 25
  - *domain* - 邮件发送者域名，默认为 ENV["HOSTNAME"]
  - *account* - 用户名，默认为 nil
  - *password* - 用户密码，默认为nil
  - *authtype* - 验证类型，默认为 *cram_md5*

SMTP 对象实例化方法调用了 sendmail, 参数如下：

- *source* - 一个字符串或数组或每个迭代器在任一时间中返回的任何东西。
- *sender* -一个字符串，出现在 email 的表单字段。
- *recipients* - 一个字符串或字符串数组，表示收件人的地址。

实例

以下提供了简单的Ruby脚本来发送邮件:

```
require 'net/smtp'

message = <<MESSAGE_END
From: Private Person <me@fromdomain.com>
To: A Test User <test@todomain.com>
Subject: SMTP e-mail test

This is a test e-mail message.
MESSAGE_END

Net::SMTP.start('localhost') do |smtp|
  smtp.send_message message, 'me@fromdomain.com',
                             'test@todomain.com'
end
```

在以上实例中，你已经设置了一个基本的电子邮件消息，注意正确的标题格式。一个电子邮件要要From，To和Subject，文本内容与头部信息间需要一个空行。

使用Net::SMTP连接到本地机器上的SMTP服务器，使用send_message方法来发送邮件，方法参数为发送者邮件与接收者邮件。

如果你没有运行在本机上的SMTP服务器，您可以使用Net::SMTP与远程SMTP服务器进行通信。如果使用网络邮件服务（如Hotmail或雅虎邮件），您的电子邮件提供者会为您提供发送邮件服务器的详细信息:

```
Net::SMTP.start('mail.your-domain.com')
```

以上代码将连接主机为 mail.your-domain.com，端口号为 25的邮件服务器，如果需要填写用户名密码，则代码如下：

```
Net::SMTP.start('mail.your-domain.com',
                25,
                'localhost',
                'username', 'password' :plain)
```

以上实例使用了指定的用户名密码连接到主机为 mail.your-domain.com，端口号为 25的邮件服务器。

使用 Ruby 发送 HTML 邮件

Net::SMTP同样提供了支持发送 HTML 格式的邮件。

发送电子邮件时你可以设置MIME版本，文档类型，字符集来发送HTML格式的邮件。

实例

以下实例用于发送 HTML 格式的邮件：

```
require 'net/smtp'

message = <<MESSAGE_END
From: Private Person <me@fromdomain.com>
To: A Test User <test@todomain.com>
MIME-Version: 1.0
Content-type: text/html
Subject: SMTP e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>
<h1>This is headline.</h1>
MESSAGE_END

Net::SMTP.start('localhost') do |smtp|
  smtp.send_message message, 'me@fromdomain.com',
                             'test@todomain.com'
end
```

发送带附件的邮件

如果需要发送混合内容的电子邮件，需要设置Content-type为multipart/mixed。 这样就可以在邮件中添加附件内容。

附件在传输前需要使用 **pack("m")** 函数将其内容转为 base64 格式。

实例

以下实例将发送附件为 /tmp/test.txt 的邮件：

```
require 'net/smtp'

filename = "/tmp/test.txt"
# 读取文件并编码为base64格式
filecontent = File.read(filename)
encodedcontent = [filecontent].pack("m")   # base64

marker = "AUNIQUEMARKER"

body =<<EOF
This is a test email to send an attachement.
EOF

# 定义主要的头部信息
part1 =<<EOF
From: Private Person <me@fromdomain.net>
To: A Test User <test@todmain.com>
Subject: Sending Attachement
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=#{marker}
--#{marker}
EOF

# 定义消息动作
part2 =<<EOF
Content-Type: text/plain
Content-Transfer-Encoding:8bit

#{body}
--#{marker}
EOF

# 定义附件部分
part3 =<<EOF
Content-Type: multipart/mixed; name=\"#{filename}\"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename="#{filename}"

#{encodedcontent}
--#{marker}--
EOF

mailtext = part1 + part2 + part3

# 发送邮件
begin
  Net::SMTP.start('localhost') do |smtp|
     smtp.sendmail(mailtext, 'me@fromdomain.net',
                          ['test@todmain.com'])
  end
rescue Exception => e
  print "Exception occured: " + e
end
```

注意：你可以指定多个发送的地址，但需要使用逗号隔开。

Ruby Socket 编程

Ruby提供了两个级别访问网络的服务，在底层你可以访问操作系统，它可以让你实现客户端和服务器为面向连接和无连接协议的基本套接字支持。

Ruby 统一支持应用程的网络协议，如FTP、HTTP等。

不管是高层的还是底层的。ruby提供了一些基本类，让你可以使用TCP,UDP,SOCKS等很多协议交互，而不必拘泥在网络层。这些类也提供了辅助类，让你可以轻松的对服务器进行读写。

接下来就让我们来学习如何进行 Ruby Socket 编程

什么是 Sockets

应用层通过传输层进行数据通信时，TCP和UDP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要 通过同一个TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了称为套接字 (Socket)的接口，区分不同应用程序进程间的网络通信和连接。

生成套接字，主要有3个参数：通信的目的IP地址、使用的传输 层协议(TCP或UDP)和使用的端口号。Socket原意是"插座"。通过将这3个参数结合起来，与一个"插座"Socket绑定，应用层就可以和传输 层通过套接字接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

Sockets 词汇解析：

选项	描述
domain	指明所使用的协议族，通常为 PF_INET, PF_UNIX, PF_X25, 等等。
type	指定socket的类型：SOCK_STREAM 或SOCK_DGRAM，Socket接口还定义了原始Socket（SOCK_RAW），允许程序使用低层协议
protocol	通常赋值0。
hostname	网络接口的标识符： 字符串, 可以是主机名或IP地址。字符串 " `<broadcast>` ", 指定 INADDR_BROADCAST 地址。 0 长度的字符串, 指定 `INADDR_ANY *` 。 一个整数，解释为主机字节顺序的二进制地址。
port	port是端口的编号，每个服务器都会监听客户端连接的一个或多个端口号，一个端口号可以是 Fixnum 的端口号, 包含了服务器名和端口。

简单的客户端

以下我们通过给定的主机和端口编写了一个简单的客户端实例，Ruby TCPSocket 类提供了 open 方法来打开一个 socke。

TCPSocket.open(hosname, port) 打开一个 TCP 连接。

一旦你打开一个 Socket 连接，你可以像 IO 对象一样读取它，完成后，你需要像关闭文件一样关闭该连接。

以下实例演示了如何连接到一个指定的主机，并从 socket 中读取数据，最后关闭socket：

```
require 'socket'      # Sockets 是标准库

hostname = 'localhost'
port = 2000

s = TCPSocket.open(hostname, port)

while line = s.gets   # 从 socket 中读取每行数据
  puts line.chop      # 打印到终端
end
s.close               # 关闭 socket
```

简单的服务

Ruby 中可以使用 TCPServer 类来写个简单的服务。TCPServer 对象是 TCPSocket 的工厂对象。

现在我们使用 TCPServer.open(hostname, port) 来创建一个 TCPServer 对象。

接下来调用 TCPServer 的 accept 方法，该方法会等到一个客户端连接到指定的端口，然后返回一个的TCPSocket对象，表示连接到该客户端。

```
require 'socket'               # 获取socket标准库

server = TCPServer.open(2000)  # Socket 监听端口为 2000
loop {                         # 永久运行服务
  client = server.accept       # 等待客户端连接
  client.puts(Time.now.ctime)  # 发送时间到客户端
  client.puts "Closing the connection. Bye!"
  client.close                 # 关闭客户端连接
}
```

现在，在服务器上运行以上代码，查看效果。

多客户端TCP服务

互联网上，大多服务都有大量的客户端连接。

Ruby的Thread类可以很容易地创建多线程服务，一个线程执行客户端的连接，而主线程在等待更多的连接。

```
require 'socket'               # 获取socket标准库

server = TCPServer.open(2000)  # Socket 监听端口为 2000
loop {                         # 永久运行服务
  Thread.start(server.accept) do |client|
    client.puts(Time.now.ctime) # 发送时间到客户端
    client.puts "Closing the connection. Bye!"
    client.close                # 关闭客户端连接
  end
}
```

在这个例子中，socket永久运行，而当server.accept接收到客户端的连接时，一个新的线程被创建并立即开始处理请求。而主程序立即循环回，并等待新的连接。

微小的Web浏览器

我们可以使用socket库来实现任何的 Internet 协议。以下代码展示了如何获取网页的内容：

```
require 'socket'

host = 'www.w3cschool.cc'     # web服务器
port = 80                           # 默认 HTTP 端口
path = "/index.htm"                 # 想要获取的文件地址

# 这是个 HTTP 请求
request = "GET #{path} HTTP/1.0\r\n\r\n"

socket = TCPSocket.open(host,port)  # 连接服务器
socket.print(request)               # 发送请求
response = socket.read              # 读取完整的响应
# Split response at first blank line into headers and body
headers,body = response.split("\r\n\r\n", 2)
print body                          # 输出结果
```

要实现一个类似 web 的客户端，你可以使用为 HTTP 预先构建的库如Net::HTTP。

以下代码与先前代码是等效的：

```
require 'net/http'                  # 我们需要的库
host = 'www.w3cschool.cc'           #  web 服务器
path = '/index.htm'                 # 我们想要的文件

http = Net::HTTP.new(host)          # 创建连接
headers, body = http.get(path)      # 请求文件
if headers.code == "200"            # 检测状态码
  print body
else
  puts "#{headers.code} #{headers.message}"
end
```

以上我们只是简单的为大家介绍 Ruby 中socket的应用，更多文档请查看：Ruby Socket 库和类方法

Ruby XML, XSLT 和 XPath 教程

什么是 XML ?

XML 指可扩展标记语言（eXtensible Markup Language）。

可扩展标记语言，标准通用标记语言的子集，一种用于标记电子文件使其具有结构性的标记语言。

它可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。 它非常适合万维网传输，提供统一的方法来描述和交换独立于应用程序或供应商的结构化数据。

更多内容请查看我们的 XML 教程

XML解析器结构和API

XML的解析器主要有DOM和SAX两种。

- SAX解析器是基于事件处理的，需要从头到尾把XML文档扫描一遍，在扫描的过程中，每次遇到一个语法结构时，就会调用这个特定语法结构的事件处理程序，向应用程序发送一个事件。
- DOM是文档对象模型解析，构建文档的分层语法结构，在内存中建立DOM树，DOM树的节点以对象的形式来标识，文档解析文成以后，文档的整个DOM树都会放在内存中。

Ruby 中解析及创建 XML

RUBY中对XML的文档的解析可以使用这个库REXML库。

REXML库是ruby的一个XML工具包，是使用纯Ruby语言编写的，遵守XML1.0规范。

在Ruby1.8版本及其以后，RUBY标准库中将包含REXML。

REXML库的路径是： rexml/document

所有的方法和类都被封装到一个REXML模块内。

REXML解析器比其他的解析器有以下优点：

- 100% 由 Ruby 编写。
- 可适用于 SAX 和 DOM 解析器。
- 它是轻量级的,不到2000行代码。

- 很容易理解的方法和类。
- 基于 SAX2 API 和完整的 XPath 支持。
- 使用 Ruby 安装，而无需单独安装。

以下为实例的 XML 代码，保存为movies.xml:

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
   <type>War, Thriller</type>
   <format>DVD</format>
   <year>2003</year>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
   <type>Anime, Science Fiction</type>
   <format>DVD</format>
   <year>1989</year>
   <rating>R</rating>
   <stars>8</stars>
   <description>A schientific fiction</description>
</movie>
   <movie title="Trigun">
   <type>Anime, Action</type>
   <format>DVD</format>
   <episodes>4</episodes>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
   <type>Comedy</type>
   <format>VHS</format>
   <rating>PG</rating>
   <stars>2</stars>
   <description>Viewable boredom</description>
</movie>
</collection>
```

DOM 解析器

让我们先来解析 XML 数据，首先我们先引入 rexml/document 库，通常我们可以将 REXML 在顶级的命名空间中引入：

```
#!/usr/bin/ruby -w

require 'rexml/document'
include REXML

xmlfile = File.new("movies.xml")
xmldoc = Document.new(xmlfile)

# 获取 root 元素
root = xmldoc.root
puts "Root element : " + root.attributes["shelf"]

# 以下将输出电影标题
xmldoc.elements.each("collection/movie"){
   |e| puts "Movie Title : " + e.attributes["title"]
}

# 以下将输出所有电影类型
xmldoc.elements.each("collection/movie/type") {
   |e| puts "Movie Type : " + e.text
}

# 以下将输出所有电影描述
xmldoc.elements.each("collection/movie/description") {
   |e| puts "Movie Description : " + e.text
}
```

以上实例输出结果为：

```
Root element : New Arrivals
Movie Title : Enemy Behind
Movie Title : Transformers
Movie Title : Trigun
Movie Title : Ishtar
Movie Type : War, Thriller
Movie Type : Anime, Science Fiction
Movie Type : Anime, Action
Movie Type : Comedy
Movie Description : Talk about a US-Japan war
Movie Description : A schientific fiction
Movie Description : Vash the Stampede!
Movie Description : Viewable boredom
SAX-like Parsing:
```

SAX 解析器

处理相同的数据文件：movies.xml，不建议SAX的解析为一个小文件，以下是个简单的实例：

```
#!/usr/bin/ruby -w

require 'rexml/document'
require 'rexml/streamlistener'
include REXML

class MyListener
  include REXML::StreamListener
  def tag_start(*args)
    puts "tag_start: #{args.map {|x| x.inspect}.join(', ')}"
  end

  def text(data)
    return if data =~ /^\w*$/     # whitespace only
    abbrev = data[0..40] + (data.length > 40 ? "..." : "")
    puts "  text   :   #{abbrev.inspect}"
  end
end

list = MyListener.new
xmlfile = File.new("movies.xml")
Document.parse_stream(xmlfile, list)
```

以上输出结果为：

```
tag_start: "collection", {"shelf"=>"New Arrivals"}
tag_start: "movie", {"title"=>"Enemy Behind"}
tag_start: "type", {}
  text   :   "War, Thriller"
tag_start: "format", {}
tag_start: "year", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "Talk about a US-Japan war"
tag_start: "movie", {"title"=>"Transformers"}
tag_start: "type", {}
  text   :   "Anime, Science Fiction"
tag_start: "format", {}
tag_start: "year", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "A schientific fiction"
tag_start: "movie", {"title"=>"Trigun"}
tag_start: "type", {}
  text   :   "Anime, Action"
tag_start: "format", {}
tag_start: "episodes", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "Vash the Stampede!"
tag_start: "movie", {"title"=>"Ishtar"}
tag_start: "type", {}
tag_start: "format", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "Viewable boredom"
```

XPath 和 Ruby

我们可以使用XPath来查看XML ,XPath 是一门在 XML 文档中查找信息的语言(查看：XPath 教程)。

XPath即为XML路径语言，它是一种用来确定XML（标准通用标记语言的子集）文档中某部分位置的语言。XPath基于XML的树状结构，提供在数据结构树中找寻节点的能力。

Ruby 通过 REXML 的 XPath 类支持 XPath，它是基于树的分析（文档对象模型）。

```
#!/usr/bin/ruby -w

require 'rexml/document'
include REXML

xmlfile = File.new("movies.xml")
xmldoc = Document.new(xmlfile)

# 第一个电影的信息
movie = XPath.first(xmldoc, "//movie")
p movie

# 打印所有电影类型
XPath.each(xmldoc, "//type") { |e| puts e.text }

# 获取所有电影格式的类型，返回数组
names = XPath.match(xmldoc, "//format").map {|x| x.text }
p names
```

以上实例输出结果为：

```
<movie title='Enemy Behind'> ... </>
War, Thriller
Anime, Science Fiction
Anime, Action
Comedy
["DVD", "DVD", "DVD", "VHS"]
```

XSLT 和 Ruby

Ruby 中有两个 XSLT 解析器，以下给出简要描述：

Ruby-Sablotron

这个解析器是由正义Masayoshi Takahash编写和维护。这主要是为Linux操作系统编写的，需要以下库：

- Sablot
- Iconv
- Expat

你可以在 Ruby-Sablotron 找到这些库。

XSLT4R

XSLT4R 由 Michael Neumann 编写。 XSLT4R 用于简单的命令行交互，可以被第三方应用程序用来转换XML文档。

XSLT4R需要XMLScan操作，包含了 XSLT4R 归档，它是一个100%的Ruby的模块。这些模块可以使用标准的Ruby安装方法（即Ruby install.rb）进行安装。

XSLT4R 语法格式如下：

```
ruby xslt.rb stylesheet.xsl document.xml [arguments]
```

如果您想在应用程序中使用XSLT4R，您可以引入XSLT及输入你所需要的参数。实例如下：

```
require "xslt"

stylesheet = File.readlines("stylesheet.xsl").to_s
xml_doc = File.readlines("document.xml").to_s
arguments = { 'image_dir' => '/....' }

sheet = XSLT::Stylesheet.new( stylesheet, arguments )

# output to StdOut
sheet.apply( xml_doc )

# output to 'str'
str = ""
sheet.output = [ str ]
sheet.apply( xml_doc )
```

更多资料

- 完整的 REXML 解析器, 请查看文档 REXML 解析器文档。
- 你可以从 RAA 知识库 中下载 XSLT4R 。

Ruby Web Services 应用 - SOAP4R

什么是 SOAP？

简单对象访问协议(SOAP,全写为Simple Object Access Protocol)是交换数据的一种协议规范。

SOAP 是一种简单的基于 XML 的协议，它使应用程序通过 HTTP 来交换信息。

简单对象访问协议是交换数据的一种协议规范，是一种轻量的、简单的、基于XML（标准通用标记语言下的一个子集）的协议，它被设计成在WEB上交换结构化的和固化的信息。

更多 SOAP 教程请查看：http://www.w3cschool.cc/soap/soap-tutorial.html。

SOAP4R 安装

SOAP4R 由Hiroshi Nakamura开发实现，用于 Ruby 的 SOAP 应用。

SOAP4R 下载地址：http://raa.ruby-lang.org/project/soap4r/。

注意：你的ruby环境可能已经安装了该该组件。

Linux 环境下你也可以使用 gem 来安装该组件，命令如下：

```
$ gem install soap4r --include-dependencies
```

如果你是window环境下开发，你需要下载zip压缩文件，并通过执行 install.rb 来安装。

SOAP4R 服务

SOAP4R 支持两种不同的服务类型：

- 基于 CGI/FastCGI 服务 (SOAP::RPC::CGIStub)
- 独立服务 (SOAP::RPC:StandaloneServer)

本教程将为大家介绍如何建立独立的 SOAP 服务。步骤如下：

第1步 - 继承SOAP::RPC::StandaloneServer

为了实现自己的独立的服务器，你需要编写一个新的类，该类为 SOAP::RPC::StandaloneServer 的子类：

```
class MyServer < SOAP::RPC::StandaloneServer
  ...............
end
```

注意：如果你要编写一个基于FastCGI的服务器，那么你需要继承 SOAP::RPC::CGIStub 类，程序的其余部分将保持不变。

第二步 - 定义处理方法

接下来我们定义Web Services的方法，如下我们定义两个方法，一个是两个数相加，一个是两个数相除：

```
class MyServer < SOAP::RPC::StandaloneServer
   ...............

   # 处理方法
   def add(a, b)
      return a + b
   end
   def div(a, b)
      return a / b
   end
end
```

第三步 - 公布处理方法

接下来添加我们在服务器上定义的方法，initialize方法是公开的，用于外部的连接：

```
class MyServer < SOAP::RPC::StandaloneServer
   def initialize(*args)
      add_method(receiver, methodName, *paramArg)
   end
end
```

以下是各参数的说明：

参数	描述
receiver	包含方法名的方法的对象。 如果你在同一个类中定义服务方法，该参数为 *self*。
methodName	调用 RPC 请求的方法名。
paramArg	参数名和参数模式

为了理解 *inout* 和 *out* 参数，考虑以下服务方法，需要输入两个参数:inParam 和 inoutParam，函数执行完成后返回三个值：retVal、inoutParam 、outParam:

```
def aMeth(inParam, inoutParam)
   retVal = inParam + inoutParam
   outParam = inParam . inoutParam
   inoutParam = inParam * inoutParam
   return retVal, inoutParam, outParam
end
```

公开的调用方法如下：

```
add_method(self, 'aMeth', [
    %w(in inParam),
    %w(inout inoutParam),
    %w(out outParam),
    %w(retval return)
])
```

第四步 - 开启服务

最后我们通过实例化派生类，并调用 start 方法来启动服务：

```
myServer = MyServer.new('ServerName',
                        'urn:ruby:ServiceName', hostname, port)

myServer.start
```

以下是请求参数的说明：

参数	描述
ServerName	服务名，你可以取你喜欢的
urn:ruby:ServiceName	Here *urn:ruby* 是固定的，但是你可以为你的服务取一个唯一的 *ServiceName*
hostname	指定主机名
port	web 服务端口

实例

接下来我们通过以上的步骤，创建一个独立的服务：

```
require "soap/rpc/standaloneserver"

begin
   class MyServer < SOAP::RPC::StandaloneServer

      # Expose our services
      def initialize(*args)
         add_method(self, 'add', 'a', 'b')
         add_method(self, 'div', 'a', 'b')
      end

      # Handler methods
      def add(a, b)
         return a + b
      end
      def div(a, b)
         return a / b
      end
  end
  server = MyServer.new("MyServer",
            'urn:ruby:calculation', 'localhost', 8080)
  trap('INT){
     server.shutdown
  }
  server.start
rescue => err
  puts err.message
end
```

执行以上程序后，就启动了一个监听 8080 端口的本地服务，并公开两个方法：add 和 div。

你可以再后台执行以上服务：

```
$ ruby MyServer.rb&
```

SOAP4R 客户端

ruby 中使用 SOAP::RPC::Driver 类开发 SOAP 客户端。接下来我们来详细看下 SOAP::RPC::Driver 类的使用。

调用 SOAP 服务需要以下信息：

- SOAP 服务 URL 地址 (SOAP Endpoint URL)
- 服务方法的命名空间（Method Namespace URI）
- 服务方法名及参数信息

接下来我们就一步步来创建 SOAP 客户端来调用以上的 SOAP 方法：add 、 div:

第一步 - 创建 SOAP Driver 实例

我们可以通过实例化 SOAP::RPC::Driver 类来调用它的新方法，如下所示：

```
SOAP::RPC::Driver.new(endPoint, nameSpace, soapAction)
```

以下是参数的描述：

参数	描述
endPoint	连接 SOAP 服务的 URL 地址
nameSpace	命名空间用于 SOAP::RPC::Driver 对象的所有 RPC .
soapAction	用于 HTTP 头部的 SOAPAction 字段值。如果是字符串是"" 则默认为 *nil*

第二步 - 添加服务方法

为 SOAP::RPC::Driver 添加 SOAP 服务方法，我们可以通过实例 SOAP::RPC::Driver 来调用以下方法：

```
driver.add_method(name, *paramArg)
```

以下是参数的说明：

参数	描述
name	远程web服务的方法名
paramArg	指定远程程序的参数

第三步 - 调用SOAP服务

最后我们可以使用 SOAP::RPC::Driver 实例来调用 SOAP 服务：

```
result = driver.serviceMethod(paramArg...)
```

serviceMethod SOAP服务的实际方法名，paramArg为方法的参数列表。

实例

基于以上的步骤，我们可以编写以下的 SOAP 客户端：

```
#!/usr/bin/ruby -w

require 'soap/rpc/driver'

NAMESPACE = 'urn:ruby:calculation'
URL = 'http://localhost:8080/'

begin
   driver = SOAP::RPC::Driver.new(URL, NAMESPACE)

   # Add remote sevice methods
   driver.add_method('add', 'a', 'b')

   # Call remote service methods
   puts driver.add(20, 30)
rescue => err
   puts err.message
end
```

以上我们只是简单介绍 Ruby 的 Web Services 。 如果你想了解更多可以查看官方文档：
Ruby 的 Web Services

Ruby 多线程

每个正在系统上运行的程序都是一个进程。每个进程包含一到多个线程。

线程是程序中一个单一的顺序控制流程，在单个程序中同时运行多个线程完成不同的工作,称为多线程。

Ruby 中我们可以通过 Thread 类来创建多线程，Ruby的线程是一个轻量级的，可以以高效的方式来实现并行的代码。

创建 Ruby 线程

要启动一个新的线程，只需要调用 Thread.new 即可:

```
# 线程 #1 代码部分
Thread.new {
  # 线程 #2 执行代码
}
# 线程 #1 执行代码
```

实例

以下实例展示了如何在Ruby程序中使用多线程：

```
#!/usr/bin/ruby

def func1
   i=0
   while i<=2
      puts "func1 at: #{Time.now}"
      sleep(2)
      i=i+1
   end
end

def func2
   j=0
   while j<=2
      puts "func2 at: #{Time.now}"
      sleep(1)
      j=j+1
   end
end

puts "Started At #{Time.now}"
t1=Thread.new{func1()}
t2=Thread.new{func2()}
t1.join
t2.join
puts "End at #{Time.now}"
```

以上代码执行结果为：

```
Started At Wed May 14 08:21:54 -0700 2014
func1 at: Wed May 14 08:21:54 -0700 2014
func2 at: Wed May 14 08:21:54 -0700 2014
func2 at: Wed May 14 08:21:55 -0700 2014
func1 at: Wed May 14 08:21:56 -0700 2014
func2 at: Wed May 14 08:21:56 -0700 2014
func1 at: Wed May 14 08:21:58 -0700 2014
End at Wed May 14 08:22:00 -0700 2014
```

线程生命周期

1、线程的创建可以使用Thread.new,同样可以以同样的语法使用Thread.start 或者Thread.fork这三个方法来创建线程。

2、创建线程后无需启动，线程会自动执行。

3、Thread 类定义了一些方法来操控线程。线程执行Thread.new中的代码块。

4、线程代码块中最后一个语句是线程的值，可以通过线程的方法来调用，如果线程执行完毕，则返回线程值，否则不返回值直到线程执行完毕。

5、Thread.current 方法返回表示当前线程的对象。 Thread.main 方法返回主线程。

6、通过 Thread.Join 方法来执行线程，这个方法会挂起主线程，直到当前线程执行完毕。

线程状态

线程有5种状态：

线程状态	返回值
Runnable	run
Sleeping	Sleeping
Aborting	aborting
Terminated normally	false
Terminated with exception	nil

线程和异常

当某线程发生异常，且没有被rescue捕捉到时，该线程通常会被无警告地终止。但是，若有其它线程因为Thread#join的关系一直等待该线程的话，则等待的线程同样会被引发相同的异常。

```
begin
  t = Thread.new do
    Thread.pass    # 主线程确实在等join
    raise "unhandled exception"
  end
  t.join
rescue
  p $!  # => "unhandled exception"
end
```

使用下列3个方法，就可以让解释器在某个线程因异常而终止时中断运行。

- 启动脚本时指定**-d**选项，并以调试模时运行。
- 用 `Thread.abort_on_exception` 设置标志。
- 使用 `Thread#abort_on_exception` 对指定的线程设定标志。

当使用上述3种方法之一后，整个解释器就会被中断。

```
t = Thread.new { ... }
t.abort_on_exception = true
```

线程同步控制

在Ruby中，提供三种实现同步的方式，分别是：

1. 通过Mutex类实现线程同步

2. 监管数据交接的Queue类实现线程同步

3. 使用ConditionVariable实现同步控制

通过**Mutex**类实现线程同步

通过Mutex类实现线程同步控制，如果在多个线程钟同时需要一个程序变量，可以将这个变量部分使用lock锁定。 代码如下：

```
#encoding:gbk
require "thread"
puts "Synchronize Thread"

@num=200
@mutex=Mutex.new

def buyTicket(num)
    @mutex.lock
        if @num>=num
            @num=@num-num
            puts "you have successfully bought #{num} tickets"
        else
            puts "sorry,no enough tickets"
        end
    @mutex.unlock
end

ticket1=Thread.new 10 do
    10.times do |value|
    ticketNum=15
    buyTicket(ticketNum)
    sleep 0.01
    end
end

ticket2=Thread.new 10 do
    10.times do |value|
    ticketNum=20
    buyTicket(ticketNum)
    sleep 0.01
    end
end

sleep 1
ticket1.join
ticket2.join
```

输出结果如下：

```
Synchronize Thread
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
```

除了使用lock锁定变量，还可以使用try_lock锁定变量，还可以使用Mutex.synchronize同步对某一个变量的访问。

监管数据交接的Queue类实现线程同步

Queue类就是表示一个支持线程的队列，能够同步对队列末尾进行访问。不同的线程可以使用统一个对类，但是不用担心这个队列中的数据是否能够同步，另外使用SizedQueue类能够限制队列的长度

SizedQueue类能够非常便捷的帮助我们开发线程同步的应用程序，应为只要加入到这个队列中，就不用关心线程的同步问题。

经典的生产者消费者问题：

```
#encoding:gbk
require "thread"
puts "SizedQuee Test"

queue = Queue.new

producer = Thread.new do
    10.times do |i|
        sleep rand(i) # 让线程睡眠一段时间
        queue << i
        puts "#{i} produced"
    end
end

consumer = Thread.new do
    10.times do |i|
        value = queue.pop
        sleep rand(i/2)
        puts "consumed #{value}"
    end
end

consumer.join
```

程序的输出：

```
SizedQuee Test
0 produced
1 produced
consumed 0
2 produced
consumed 1
consumed 2
3 produced
consumed 34 produced

consumed 4
5 produced
consumed 5
6 produced
consumed 6
7 produced
consumed 7
8 produced
9 produced
consumed 8
consumed 9
```

使用ConditionVariable实现同步控制

使用 ConditonVariable进行同步控制，能够在一些致命的资源竞争部分挂起线程直到有可用的资源为止。

```
#encoding:gbk
require "thread"
puts "thread synchronize by ConditionVariable"

mutex = Mutex.new
resource = ConditionVariable.new

a = Thread.new {
    mutex.synchronize {
        # 这个线程目前需要resource这个资源
        resource.wait(mutex)
        puts "get resource"
    }
}

b = Thread.new {
    mutex.synchronize {
        #线程b完成对resourece资源的使用并释放resource
        resource.signal
    }
}

a.join
puts "complete"
```

mutex 是声明的一个资源，然后通过ConditionVariable来控制申请和释放这个资源。

b 线程完成了某些工作之后释放资源resource.signal,这样a线程就可以获得一个mutex资源然后进行执行。 执行结果：

```
thread synchronize by ConditionVariable
get resource
complete
```

线程类方法

完整的 Thread（线程） 类方法如下：

方法	描述
Thread.abort_on_exception	若其值为真的话，一旦某线程因异常而终止时，整个解释器就会被中断。它的默认值是假，也就是说，在通常情况下，若某线程发生异常且该异常未被Thread#join等检测到时，该线程会被无警告地终止。
Thread.abort_on_exception=	如果设置为 *true*, 一旦某线程因异常而终止时，整个解释器就会被中断。返回新的状态
Thread.critical	返回布尔值。
Thread.critical=	当其值为true时，将不会进行线程切换。若当前线程挂起(stop)或有信号(signal)干预时，其值将自动变为false。
Thread.current	返回当前运行中的线程(当前线程)。
Thread.exit	终止当前线程的运行。返回当前线程。若当前线程是唯一的一个线程时，将使用exit(0)来终止它的运行。
Thread.fork { block }	与 Thread.new 一样生成线程。
Thread.kill(aThread)	终止线程的运行.
Thread.list	返回处于运行状态或挂起状态的活线程的数组。
Thread.main	返回主线程。
Thread.new([arg]*) {| args | block }	生成线程,并开始执行。数会被原封不动地传递给块. 这就可以在启动线程的同时,将值传递给该线程所固有的局部变量。
Thread.pass	将运行权交给其他线程. 它不会改变运行中的线程的状态,而是将控制权交给其他可运行的线程(显式的线程调度)。
Thread.start([args]*) {| args | block }	生成线程,并开始执行。数会被原封不动地传递给块. 这就可以在启动线程的同时,将值传递给该线程所固有的局部变量。
Thread.stop	将当前线程挂起,直到其他线程使用run方法再次唤醒该线程。

线程实例化方法

以下实例调用了线程实例化方法 join：

```
#!/usr/bin/ruby

thr = Thread.new do   # 实例化
   puts "In second thread"
   raise "Raise exception"
end
thr.join   # 调用实例化方法 join
```

以下是完整实例化方法列表：

序号	方法描述
thr[name]	取出线程内与name相对应的固有数据。 name可以是字符串或符号。 若没有与name相对应的数据时, 返回nil。
thr[name] =	设置线程内name相对应的固有数据的值， name可以是字符串或符号。 若设为nil时, 将删除该线程内对应数据。
thr.abort_on_exception	返回布尔值。
thr.abort_on_exception=	若其值为true的话， 一旦某线程因异常而终止时， 整个解释器就会被中断。
thr.alive?	若线程是"活"的,就返回true。
thr.exit	终止线程的运行。返回self。
thr.join	挂起当前线程,直到self线程终止运行为止. 若self因异常而终止时, 将会当前线程引发同样的异常。
thr.key?	若与name相对应的线程固有数据已经被定义的话,就返回true
thr.kill	类似于 *Thread.exit* 。
thr.priority	返回线程的优先度. 优先度的默认值为0. 该值越大则优先度越高.
thr.priority=	设定线程的优先度. 也可以将其设定为负数.
thr.raise(anException)	在该线程内强行引发异常.
thr.run	重新启动被挂起(stop)的线程. 与wakeup不同的是,它将立即进行线程的切换. 若对死进程使用该方法时, 将引发ThreadError异常.
thr.safe_level	返回self 的安全等级. 当前线程的safe_level与$SAFE相同.
thr.status	使用字符串"run"、"sleep"或"aborting" 来表示活线程的状态. 若某线程是正常终止的话,就返回false. 若因异常而终止的话,就返回nil。
thr.stop?	若线程处于终止状态(dead)或被挂起(stop)时,返回true.
thr.value	一直等到self线程终止运行(等同于join)后,返回该线程的块的返回值. 若在线程的运行过程中发生了异常, 就会再次引发该异常.
thr.wakeup	把被挂起(stop)的线程的状态改为可执行状态(run), 若对死线程执行该方法时,将会引发ThreadError异常。

W3School Java 教程

来源：Java教程

整理：飞龙

Java 基础

Java 简介

Java是由Sun Microsystems公司于1995年5月推出的Java面向对象程序设计语言和Java平台的总称。由James Gosling和同事们共同研发，并在1995年正式推出。

Java分为三个体系：

- JavaSE（J2SE）（Java2 Platform Standard Edition，java平台标准版）
- JavaEE(J2EE)(Java 2 Platform,Enterprise Edition，java平台企业版)
- JavaME(J2ME)(Java 2 Platform Micro Edition，java平台微型版)。

2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名以取消其中的数字"2"：J2EE更名为Java EE, J2SE更名为Java SE，J2ME更名为Java ME。

主要特性

- **Java语言是简单的：**

  Java语言的语法与C语言和C++语言很接近，使得大多数程序员很容易学习和使用。另一方面，Java丢弃了C++中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。特别地，Java语言不使用指针，而是引用。并提供了自动的废料收集，使得程序员不必为内存管理而担忧。

- **Java语言是面向对象的：**

  Java语言提供类、接口和继承等原语，为了简单起见，只支持类之间的单继承，但支持接口之间的多继承，并支持类与接口之间的实现机制（关键字为implements）。Java语言全面支持动态绑定，而C++语言只对虚函数使用动态绑定。总之，Java语言是一个纯的面向对象程序设计语言。

- **Java语言是分布式的：**

  Java语言支持Internet应用的开发，在基本的Java应用编程接口中有一个网络应用编程接口（java net），它提供了用于网络应用编程的类库，包括URL、URLConnection、Socket、ServerSocket等。Java的RMI（远程方法激活）机制也是开发分布式应用的重要手段。

- **Java语言是健壮的：**

  Java的强类型机制、异常处理、垃圾的自动收集等是Java程序健壮性的重要保证。对指针的丢弃是Java的明智选择。Java的安全检查机制使得Java更具健壮性。

- **Java语言是安全的：**

Java通常被用在网络环境中，为此，Java提供了一个安全机制以防恶意代码的攻击。除了Java语言具有的许多安全特性以外，Java对通过网络下载的类具有一个安全防范机制（类ClassLoader），如分配不同的名字空间以防替代本地的同名类、字节代码检查，并提供安全管理机制（类SecurityManager）让Java应用设置安全哨兵。

- **Java**语言是体系结构中立的：

  Java程序（后缀为java的文件）在Java平台上被编译为体系结构中立的字节码格式（后缀为class的文件），然后可以在实现这个Java平台的任何系统中运行。这种途径适合于异构的网络环境和软件的分发。

- **Java**语言是可移植的：

  这种可移植性来源于体系结构中立性，另外，Java还严格规定了各个基本数据类型的长度。Java系统本身也具有很强的可移植性，Java编译器是用Java实现的，Java的运行环境是用ANSI C实现的。

- **Java**语言是解释型的：

  如前所述，Java程序在Java平台上被编译为字节码格式，然后可以在实现这个Java平台的任何系统中运行。在运行时，Java平台中的Java解释器对这些字节码进行解释执行，执行过程中需要的类在联接阶段被载入到运行环境中。

- **Java**是高性能的：

  与那些解释型的高级脚本语言相比，Java的确是高性能的。事实上，Java的运行速度随着JIT(Just-In-Time）编译器技术的发展越来越接近于C++。

- **Java**语言是多线程的：

  在Java语言中，线程是一种特殊的对象，它必须由Thread类或其子（孙）类来创建。通常有两种方法来创建线程：其一，使用型构为Thread(Runnable)的构造子将一个实现了Runnable接口的对象包装成一个线程，其二，从Thread类派生出子类并重写run方法，使用该子类创建的对象即为线程。值得注意的是Thread类已经实现了Runnable接口，因此，任何一个线程均有它的run方法，而run方法中包含了线程所要运行的代码。线程的活动由一组方法来控制。Java语言支持多个线程的同时执行，并提供多线程之间的同步机制（关键字为synchronized）。

- **Java**语言是动态的：

  Java语言的设计目标之一是适应于动态变化的环境。Java程序需要的类能够动态地被载入到运行环境，也可以通过网络来载入所需要的类。这也有利于软件的升级。另外，Java中的类有一个运行时刻的表示，能进行运行时刻的类型检查。

发展历史

- 1995年5月23日，Java语言诞生
- 1996年1月，第一个JDK-JDK1.0诞生
- 1996年4月，10个最主要的操作系统供应商申明将在其产品中嵌入JAVA技术
- 1996年9月，约8.3万个网页应用了JAVA技术来制作
- 1997年2月18日，JDK1.1发布
- 1997年4月2日，JavaOne会议召开，参与者逾一万人，创当时全球同类会议规模之纪录
- 1997年9月，JavaDeveloperConnection社区成员超过十万
- 1998年2月，JDK1.1被下载超过2,000,000次
- 1998年12月8日，JAVA2企业平台J2EE发布
- 1999年6月，SUN公司发布Java的三个版本：标准版（JavaSE,以前是J2SE）、企业版（JavaEE以前是J2EE）和微型版（JavaME，以前是J2ME）
- 2000年5月8日，JDK1.3发布
- 2000年5月29日，JDK1.4发布
- 2001年6月5日，NOKIA宣布，到2003年将出售1亿部支持Java的手机
- 2001年9月24日，J2EE1.3发布
- 2002年2月26日，J2SE1.4发布，自此Java的计算能力有了大幅提升
- 2004年9月30日18:00PM，J2SE1.5发布，成为Java语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5更名为Java SE 5.0
- 2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名，以取消其中的数字"2"：J2EE更名为Java EE，J2SE更名为Java SE，J2ME更名为Java ME
- 2006年12月，SUN公司发布JRE6.0
- 2009年04月20日，甲骨文74亿美元收购Sun。取得java的版权。
- 2010年11月，由于甲骨文对于Java社区的不友善，因此Apache扬言将退出JCP[4]。
- 2011年7月28日，甲骨文发布java7.0的正式版。

Java开发工具

Java语言尽量保证系统内存在1G以上，其他工具如下所示：

- Linux 系统或者Windows 95/98/2000/XP，WIN 7/8系统
- Java JDK 7
- Notepad编辑器或者其他编辑器。
- IDE：Eclipse

安装好以上的工具后，我们就可以输出Java的第一个程序"Hello World！"

```
public class MyFirstJavaProgram {

    public static void main(String []args) {
       System.out.println("Hello World");
    }
}
```

在下一章节我们将介绍如何配置java开发环境。

Java开发环境配置

在本章节中我们将为大家介绍如何搭建Java开发环境。

window系统安装java

下载JDK

首先我们需要下载java开发工具包JDK，下载地址：http://www.oracle.com/technetwork/java/javase/downloads/index.html，点击如下下载按钮：

下载后JDK的安装根据提示进行，还有安装JDK的时候也会安装JRE，一并安装就可以了。

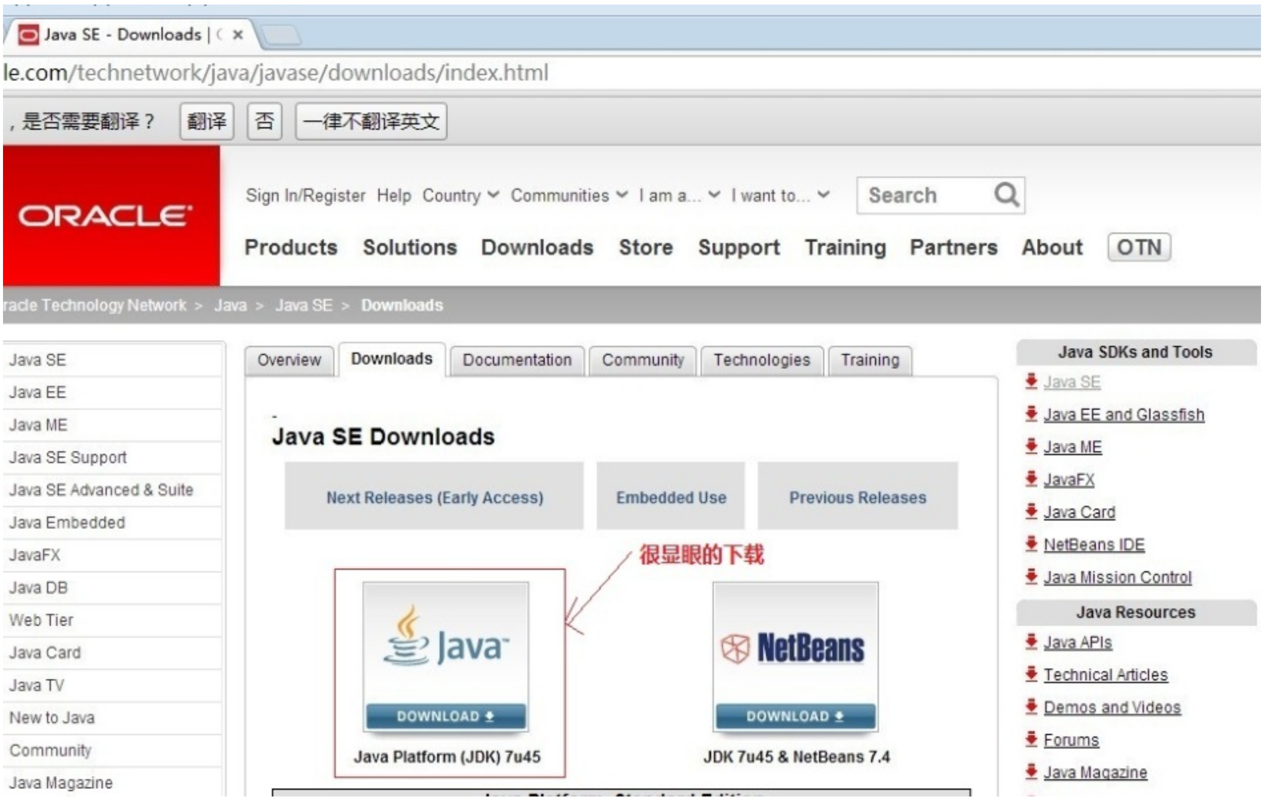
安装JDK，安装过程中可以自定义安装目录等信息，例如我们选择安装目录为C:\Program Files\Java\jdk1.7.0 。

配置环境变量

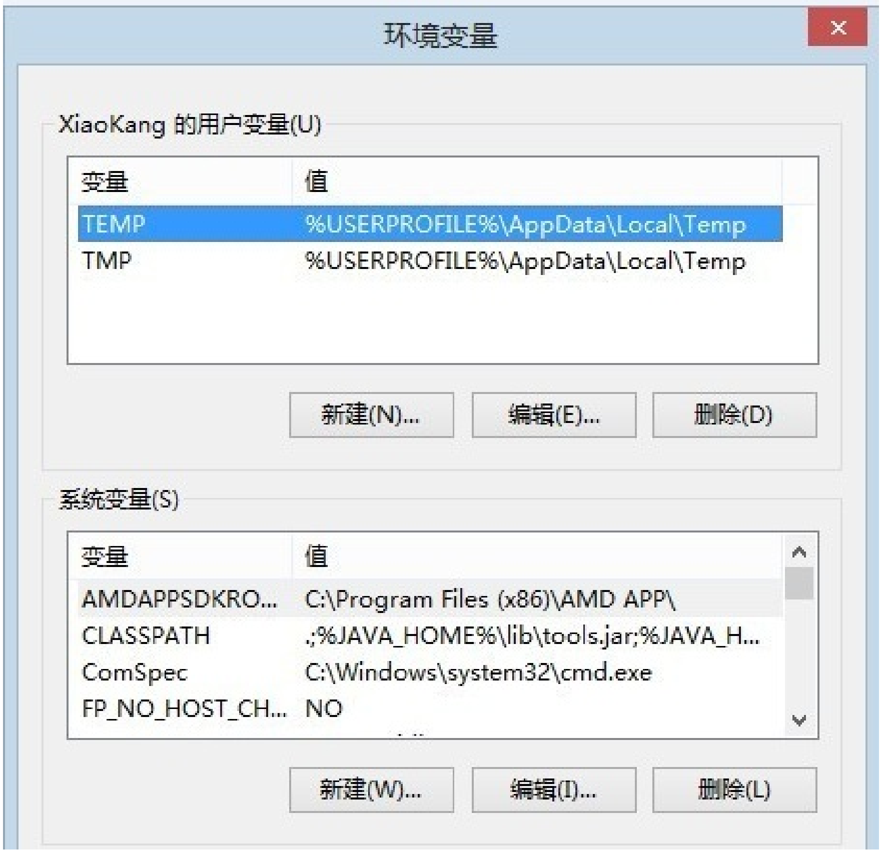
1.安装完成后，右击"我的电脑"，点击"属性"；

2.选择"高级"选项卡，点击"环境变量"；

然后就会出现如下图所示的画面



在"系统变量"中设置3项属性，JAVA_HOME,PATH,CLASSPATH(大小写无所谓),若已存在则点击"编辑"，不存在则点击"新建"。



变量设置

- 变量名：JAVA_HOME

- 变量值：C:\Program Files\Java\jdk1.7.0

//这里是你JDK的安装路径，可以更换

- 变量名：CLASSPATH
- 变量值：.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar; //记得前面有个"."

- 变量名：Path

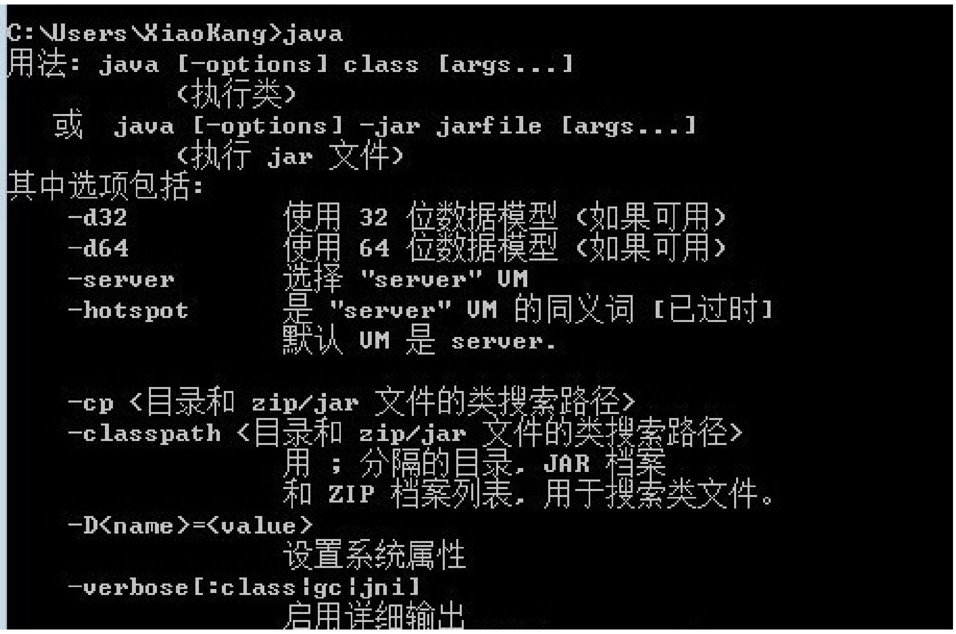
- 变量值：%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

这是java的环境配置，配置完成后直接启动eclipse，它会自动完成java环境的配置。

测试JDK是否安装成功

1、"开始"->;"运行"，键入"cmd"；

2、键入命令"java -version"，"java"，"javac"几个命令，出现画面，说明环境变量配置成功；



```
C:\Users\XiaoKang>java
用法: java [-options] class [args...]
           (执行类)
   或  java [-options] -jar jarfile [args...]
           (执行 jar 文件)
其中选项包括:
    -d32          使用 32 位数据模型 (如果可用)
    -d64          使用 64 位数据模型 (如果可用)
    -server       选择 "server" VM
    -hotspot      是 "server" VM 的同义词 [已过时]
                  默认 VM 是 server.

    -cp <目录和 zip/jar 文件的类搜索路径>
    -classpath <目录和 zip/jar 文件的类搜索路径>
                  用 ; 分隔的目录, JAR 档案
                  和 ZIP 档案列表, 用于搜索类文件。
    -D<name>=<value>
                  设置系统属性
    -verbose[:class|gc|jni]
                  启用详细输出
```

Linux，UNIX，Solaris，FreeBSD环境变量设置

环境变量PATH应该设定为指向Java二进制文件安装的位置。如果设置遇到困难，请参考shell文档。

例如，假设你使用bash作为shell，你可以把下面的内容添加到你的 .bashrc文件结尾: export PATH=/path/to/java:$PATH

流行JAVA开发工具

正所谓工欲善其事必先利其器，我们在开发java语言过程中同样需要依款不错的开发工具，目前市场上的IDE很多，本文为大家推荐一下几款java开发工具：

- **Notepad++ :** Notepad++ 是在微软视窗环境之下的一个免费的代码编辑器，下载地址：http://notepad-plus-plus.org/
- **Netbeans:**开源免费的java IDE，下载地址：http://www.netbeans.org/index.html
- **Eclipse:**另一个免费开源的java IDE，下载地址：http://www.eclipse.org/

Java基础语法

一个Java程序可以认为是一系列对象的集合，而这些对象通过调用彼此的方法来协同工作。下面简要介绍下类、对象、方法和实例变量的概念。

- 对象：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- 类：类是一个模板，它描述一类对象的行为和状态。
- 方法：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。
- 实例变量：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。

第一个Java程序

下面看一个简单的Java程序，它将打印字符串 *Hello World*

```
public class MyFirstJavaProgram {
   /* 第一个Java程序.
    * 它将打印字符串 Hello World
    */
    public static void main(String []args) {
       System.out.println("Hello World"); // 打印 Hello World
    }
}
```

下面将逐步介绍如何保存、编译以及运行这个程序：

- 打开Notepad，把上面的代码添加进去；
- 把文件名保存为：MyFirstJavaProgram.java；
- 打开cmd命令窗口，进入目标文件所在的位置，假设是C:\
- 在命令行窗口键入 javac MyFirstJavaProgram.java 按下enter键编译代码。如果代码没有错误，cmd命令提示符会进入下一行。（假设环境变量都设置好了）。
- 再键入java MyFirstJavaProgram 按下Enter键就可以运行程序了

你将会在窗口看到 Hello World

```
C : > javac MyFirstJavaProgram.java
C : > java MyFirstJavaProgram
Hello World
```

基本语法

编写Java程序时，应注意以下几点：

- 大小写敏感：Java是大小写敏感的，这就意味着标识符Hello与hello是不同的。
- 类名：对于所有的类来说，类名的首字母应该大写。如果类名由若干单词组成，那么每个单词的首字母应该大写，例如 MyFirstJavaClass 。
- 方法名：所有的方法名都应该以小写字母开头。如果方法名含有若干单词，则后面的每个单词首字母大写。
- 源文件名：源文件名必须和类名相同。当保存文件的时候，你应该使用类名作为文件名保存（切记Java是大小写敏感的），文件名的后缀为.java。（如果文件名和类名不相同则会导致编译错误）。
- 主方法入口：所有的Java 程序由**public static void main(String args[])**方法开始执行。

Java标识符

Java所有的组成部分都需要名字。类名、变量名以及方法名都被称为标识符。

关于Java标识符，有以下几点需要注意：

- 所有的标识符都应该以字母（A-Z或者a-z）,美元符（$）、或者下划线（_）开始
- 首字符之后可以是任何字符的组合
- 关键字不能用作标识符
- 标识符是大小写敏感的
- 合法标识符举例：age、$salary、_value、__1_value
- 非法标识符举例：123abc、-salary

Java修饰符

像其他语言一样，Java可以使用修饰符来修饰类中方法和属性。主要有两类修饰符：

- 可访问修饰符 : default, public , protected, private
- 不可访问修饰符 : final, abstract, strictfp

在后面的章节中我们会深入讨论Java修饰符。

Java变量

Java中主要有如下几种类型的变量

- 局部变量
- 类变量（静态变量）
- 成员变量（非静态变量）

Java数组

数组是储存在堆上的对象，可以保存多个同类型变量。在后面的章节中，我们将会学到如何声明、构造以及初始化一个数组。

Java枚举

Java 5.0引入了枚举，枚举限制变量只能是预先设定好的值。使用枚举可以减少代码中的bug。

例如，我们为果汁店设计一个程序，它将限制果汁为小杯、中杯、大杯。这就意味着它不允许顾客点除了这三种尺寸外的果汁。

实例

```
class FreshJuice {
   enum FreshJuiceSize{ SMALL, MEDUIM, LARGE }
   FreshJuiceSize size;
}

public class FreshJuiceTest {
   public static void main(String args[]){
      FreshJuice juice = new FreshJuice();
      juice.size = FreshJuice. FreshJuiceSize.MEDUIM ;
   }
}
```

注意：枚举可以单独声明或者声明在类里面。方法、变量、构造函数也可以在枚举中定义。

Java关键字

下面列出了Java保留字。这些保留字不能用于常量、变量、和任何标识符的名称。

关键字	描述
abstract	抽象方法，抽象类的修饰符
assert	断言条件是否满足
boolean	布尔数据类型
break	跳出循环或者label代码段
byte	8-bit 有符号数据类型
case	switch语句的一个条件
catch	和try搭配扑捉异常信息
char	16-bit Unicode字符数据类型
class	定义类

const	未使用
continue	不执行循环体剩余部分
default	switch语句中的默认分支
do	循环语句，循环体至少会执行一次
double	64-bit双精度浮点数
else	if条件不成立时执行的分支
enum	枚举类型
extends	表示一个类是另一个类的子类
final	表示一个值在初始化之后就不能再改变了 表示方法不能被重写，或者一个类不能有子类
finally	为了完成执行的代码而设计的，主要是为了程序的健壮性和完整性，无论有没有异常发生都执行代码。
float	32-bit单精度浮点数
for	for循环语句
goto	未使用
if	条件语句
implements	表示一个类实现了接口
import	导入类
instanceof	测试一个对象是否是某个类的实例
int	32位整型数
interface	接口，一种抽象的类型，仅有方法和常量的定义
long	64位整型数
native	表示方法用非java代码实现
new	分配新的类实例
package	一系列相关类组成一个包
private	表示私有字段，或者方法等，只能从类内部访问
protected	表示字段只能通过类或者其子类访问 子类或者在同一个包内的其他类
public	表示共有属性或者方法
return	方法返回值
short	16位数字
static	表示在类级别定义，所有实例共享的
strictfp	浮点数比较使用严格的规则

super	表示基类
switch	选择语句
synchronized	表示同一时间只能由一个线程访问的代码块
this	表示调用当前实例 或者调用另一个构造函数
throw	抛出异常
throws	定义方法可能抛出的异常
transient	修饰不要序列化的字段
try	表示代码块要做异常处理或者和finally配合表示是否抛出异常都执行finally中的代码
void	标记方法不返回任何值
volatile	标记字段可能会被多个线程同时访问，而不做同步
while	while循环

Java注释

类似于C/C++，Java也支持单行以及多行注释。注释中的字符将被Java编译器忽略。

```
public class MyFirstJavaProgram{
   /* 这是第一个Java程序
    *它将打印Hello World
    * 这是一个多行注释的示例
    */
    public static void main(String []args){
       // 这是单行注释的示例
       /* 这个也是单行注释的示例 */
       System.out.println("Hello World");
    }
}
```

Java 空行

空白行，或者有注释的的行，Java编译器都会忽略掉。

继承

在Java中，一个类可以由其他类派生。如果你要创建一个类，而且已经存在一个类具有你所需要的属性或方法，那么你可以将新创建的类继承该类。

利用继承的方法，可以重用已存在类的方法和属性，而不用重写这些代码。被继承的类称为超类（super class），派生类称为子类（subclass）。

接口

在Java中，接口可理解为对象间相互通信的协议。接口在继承中扮演着很重要的角色。

接口只定义派生要用到的方法，但是方法的具体实现完全取决于派生类。

下一节介绍Java编程中的类和对象。之后你将会对Java中的类和对象有更清楚的认识。

Java对象和类

Java作为一种面向对象语言。支持以下基本概念：

- 多态
- 继承
- 封装
- 抽象
- 类
- 对象
- 实例
- 方法
- 消息解析

本节我们重点研究对象和类的概念。

- 对象：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- 类：类是一个模板，它描述一类对象的行为和状态。

Java中的对象

现在让我们深入了解什么是对象。看看周围真实的世界，会发现身边有很多对象，车，狗，人等等。所有这些对象都有自己的状态和行为。

拿一条狗来举例，它的状态有：名字、品种、颜色，行为有：叫、摇尾巴和跑。

对比现实对象和软件对象，它们之间十分相似。

软件对象也有状态和行为。软件对象的状态就是属性，行为通过方法体现。

在软件开发中，方法操作对象内部状态的改变，对象的相互调用也是通过方法来完成。

Java中的类

类可以看成是创建Java对象的模板。

通过下面一个简单的类来理解下Java中类的定义：

```
public class Dog{
   String breed;
   int age;
   String color;
   void barking(){
   }

   void hungry(){
   }

   void sleeping(){
   }
}
```

一个类可以包含以下类型变量：

- 局部变量：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
- 成员变量：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
- 类变量：类变量也声明在类中，方法体之外，但必须声明为static类型。

一个类可以拥有多个方法，在上面的例子中：barking()、hungry()和sleeping()都是Dog类的方法。

构造方法

每个类都有构造方法。如果没有显式地为类定义构造方法，Java编译器将会为该类提供一个默认构造方法。

在创建一个对象的时候，至少要调用一个构造方法。构造方法的名称必须与类同名，一个类可以有多个构造方法。

下面是一个构造方法示例：

```
public class Puppy{
   public Puppy(){
   }

   public Puppy(String name){
      // 这个构造器仅有一个参数：name
   }
}
```

创建对象

对象是根据类创建的。在Java中，使用关键字new来创建一个新的对象。创建对象需要以下三步：

- 声明：声明一个对象，包括对象名称和对象类型。

- 实例化：使用关键字new来创建一个对象。
- 初始化：使用new创建对象时，会调用构造方法初始化对象。

下面是一个创建对象的例子：

```
public class Puppy{
   public Puppy(String name){
      //这个构造器仅有一个参数：name
      System.out.println("Passed Name is :" + name );
   }
   public static void main(String []args){
      // 下面的语句将创建一个Puppy对象
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

编译并运行上面的程序，会打印出下面的结果：

```
Passed Name is :tommy
```

访问实例变量和方法

通过已创建的对象来访问成员变量和成员方法，如下所示：

```
/* 实例化对象 */
ObjectReference = new Constructor();
/* 访问其中的变量 */
ObjectReference.variableName;
/* 访问类中的方法 */
ObjectReference.MethodName();
```

实例

下面的例子展示如何访问实例变量和调用成员方法：

```
public class Puppy{
   int puppyAge;
   public Puppy(String name){
      // 这个构造器仅有一个参数：name
      System.out.println("Passed Name is :" + name );
   }

   public void setAge( int age ){
       puppyAge = age;
   }

   public int getAge( ){
       System.out.println("Puppy's age is :" + puppyAge );
       return puppyAge;
   }

   public static void main(String []args){
      /* 创建对象 */
      Puppy myPuppy = new Puppy( "tommy" );
      /* 通过方法来设定age */
      myPuppy.setAge( 2 );
      /* 调用另一个方法获取age */
      myPuppy.getAge( );
      /*你也可以像下面这样访问成员变量 */
      System.out.println("Variable Value :" + myPuppy.puppyAge );
   }
}
```

编译并运行上面的程序，产生如下结果：

```
Passed Name is :tommy
Puppy's age is :2
Variable Value :2
```

源文件声明规则

在本节的最后部分，我们将学习源文件的声明规则。当在一个源文件中定义多个类，并且还有import语句和package语句时，要特别注意这些规则。

- 一个源文件中只能有一个public类
- 一个源文件可以有多个非public类
- 源文件的名称应该和public类的类名保持一致。例如：源文件中public类的类名是Employee，那么源文件应该命名为Employee.java。
- 如果一个类定义在某个包中，那么package语句应该在源文件的首行。
- 如果源文件包含import语句，那么应该放在package语句和类定义之间。如果没有package语句，那么import语句应该在源文件中最前面。
- import语句和package语句对源文件中定义的所有类都有效。在同一源文件中，不能给不同的类不同的包声明。

类有若干种访问级别，并且类也分不同的类型：抽象类和final类等。这些将在访问控制章节介绍。

除了上面提到的几种类型，Java还有一些特殊的类，如：内部类、匿名类。

Java包

包主要用来对类和接口进行分类。当开发Java程序时，可能编写成百上千的类，因此很有必要对类和接口进行分类。

Import语句

在Java中，如果给出一个完整的限定名，包括包名、类名，那么Java编译器就可以很容易地定位到源代码或者类。Import语句就是用来提供一个合理的路径，使得编译器可以找到某个类。

例如，下面的命令行将会命令编译器载入java_installation/java/io路径下的所有类

```
import java.io.*;
```

一个简单的例子

在该例子中，我们创建两个类：Employee和EmployeeTest。

首先打开文本编辑器，把下面的代码粘贴进去。注意将文件保存为Employee.java。

Employee类有四个成员变量：name、age、designation和salary。该类显式声明了一个构造方法，该方法只有一个参数。

```
import java.io.*;
public class Employee{
   String name;
   int age;
   String designation;
   double salary;
   // Employee 类的构造器
   public Employee(String name){
      this.name = name;
   }
   // 设置age的值
   public void empAge(int empAge){
      age =  empAge;
   }
   /* 设置designation的值*/
   public void empDesignation(String empDesig){
      designation = empDesig;
   }
   /* 设置salary的值*/
   public void empSalary(double empSalary){
      salary = empSalary;
   }
   /* 打印信息 */
   public void printEmployee(){
      System.out.println("Name:"+ name );
      System.out.println("Age:" + age );
      System.out.println("Designation:" + designation );
      System.out.println("Salary:" + salary);
   }
}
```

程序都是从main方法开始执行。为了能运行这个程序，必须包含main方法并且创建一个实例对象。

下面给出EmployeeTest类，该类实例化2个Employee类的实例，并调用方法设置变量的值。

将下面的代码保存在EmployeeTest.java文件中。

```
import java.io.*;
public class EmployeeTest{

   public static void main(String args[]){
      /* 使用构造器创建两个对象 */
      Employee empOne = new Employee("James Smith");
      Employee empTwo = new Employee("Mary Anne");

      // 调用这两个对象的成员方法
      empOne.empAge(26);
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

      empTwo.empAge(21);
      empTwo.empDesignation("Software Engineer");
      empTwo.empSalary(500);
      empTwo.printEmployee();
   }
}
```

编译这两个文件并且运行EmployeeTest类，可以看到如下结果：

```
C :> javac Employee.java
C :> vi EmployeeTest.java
C :> javac  EmployeeTest.java
C :> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

Java基本数据类型

变量就是申请内存来存储值。也就是说，当创建变量的时候，需要在内存中申请空间。

内存管理系统根据变量的类型为变量分配存储空间，分配的空间只能用来储存该类型数据。

因此，通过定义不同类型的变量，可以在内存中储存整数、小数或者字符。

Java的两大数据类型:

- 内置数据类型
- 引用数据类型

内置数据类型

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

byte：

- byte数据类型是8位、有符号的，以二进制补码表示的整数；
- 最小值是-128（-2^7）；
- 最大值是127（2^7-1）；
- 默认值是0；
- byte类型用在大型数组中节约空间，主要代替整数，因为byte变量占用的空间只有int类型的四分之一；
- 例子：byte a = 100，byte b = -50。

short：

- short数据类型是16位、有符号的以二进制补码表示的整数
- 最小值是-32768（-2^15）；
- 最大值是32767（2^15 - 1）；
- Short数据类型也可以像byte那样节省空间。一个short变量是int型变量所占空间的二分之一；
- 默认值是0；
- 例子：short s = 1000，short r = -20000。

int：

- int数据类型是32位、有符号的以二进制补码表示的整数；
- 最小值是-2,147,483,648（-2^31）；
- 最大值是2,147,485,647（2^31 - 1）；

- 一般地整型变量默认为int类型；
- 默认值是0；
- 例子：int a = 100000, int b = -200000。

long：

- long数据类型是64位、有符号的以二进制补码表示的整数；
- 最小值是-9,223,372,036,854,775,808（-2^63）；
- 最大值是9,223,372,036,854,775,807（2^63 -1）；
- 这种类型主要使用在需要比较大整数的系统上；
- 默认值是0L；
- 例子： long a = 100000L，int b = -200000L。

float：

- float数据类型是单精度、32位、符合IEEE 754标准的浮点数；
- float在储存大型浮点数组的时候可节省内存空间；
- 默认值是0.0f；
- 浮点数不能用来表示精确的值，如货币；
- 例子：float f1 = 234.5f。

double：

- double数据类型是双精度、64位、符合IEEE 754标准的浮点数；
- 浮点数的默认类型为double类型；
- double类型同样不能表示精确的值，如货币；
- 默认值是0.0f；
- 例子：double d1 = 123.4。

boolean：

- boolean数据类型表示一位的信息；
- 只有两个取值：true和false；
- 这种类型只作为一种标志来记录true/false情况；
- 默认值是false；
- 例子：boolean one = true。

char：

- char类型是一个单一的16位Unicode字符；
- 最小值是'\u0000'（即为0）；
- 最大值是'\uffff'（即为65,535）；
- char数据类型可以储存任何字符；
- 例子：char letter = 'A'。

实例

对于数值类型的基本类型的取值范围，我们无需强制去记忆，因为它们的值都已经以常量的形式定义在对应的包装类中了。请看下面的例子：

```
public class PrimitiveTypeTest {
    public static void main(String[] args) {
        // byte
        System.out.println("基本类型：byte 二进制位数：" + Byte.SIZE);
        System.out.println("包装类：java.lang.Byte");
        System.out.println("最小值：Byte.MIN_VALUE=" + Byte.MIN_VALUE);
        System.out.println("最大值：Byte.MAX_VALUE=" + Byte.MAX_VALUE);
        System.out.println();

        // short
        System.out.println("基本类型：short 二进制位数：" + Short.SIZE);
        System.out.println("包装类：java.lang.Short");
        System.out.println("最小值：Short.MIN_VALUE=" + Short.MIN_VALUE);
        System.out.println("最大值：Short.MAX_VALUE=" + Short.MAX_VALUE);
        System.out.println();

        // int
        System.out.println("基本类型：int 二进制位数：" + Integer.SIZE);
        System.out.println("包装类：java.lang.Integer");
        System.out.println("最小值：Integer.MIN_VALUE=" + Integer.MIN_VALUE);
        System.out.println("最大值：Integer.MAX_VALUE=" + Integer.MAX_VALUE);
        System.out.println();

        // long
        System.out.println("基本类型：long 二进制位数：" + Long.SIZE);
        System.out.println("包装类：java.lang.Long");
        System.out.println("最小值：Long.MIN_VALUE=" + Long.MIN_VALUE);
        System.out.println("最大值：Long.MAX_VALUE=" + Long.MAX_VALUE);
        System.out.println();

        // float
        System.out.println("基本类型：float 二进制位数：" + Float.SIZE);
        System.out.println("包装类：java.lang.Float");
        System.out.println("最小值：Float.MIN_VALUE=" + Float.MIN_VALUE);
        System.out.println("最大值：Float.MAX_VALUE=" + Float.MAX_VALUE);
        System.out.println();

        // double
        System.out.println("基本类型：double 二进制位数：" + Double.SIZE);
        System.out.println("包装类：java.lang.Double");
        System.out.println("最小值：Double.MIN_VALUE=" + Double.MIN_VALUE);
        System.out.println("最大值：Double.MAX_VALUE=" + Double.MAX_VALUE);
        System.out.println();

        // char
        System.out.println("基本类型：char 二进制位数：" + Character.SIZE);
        System.out.println("包装类：java.lang.Character");
        // 以数值形式而不是字符形式将Character.MIN_VALUE输出到控制台
        System.out.println("最小值：Character.MIN_VALUE="
                + (int) Character.MIN_VALUE);
        // 以数值形式而不是字符形式将Character.MAX_VALUE输出到控制台
        System.out.println("最大值：Character.MAX_VALUE="
                + (int) Character.MAX_VALUE);
    }
}
```

编译以上代码输出结果如下所示：

```
基本类型：byte 二进制位数：8
包装类：java.lang.Byte
最小值：Byte.MIN_VALUE=-128
最大值：Byte.MAX_VALUE=127

基本类型：short 二进制位数：16
包装类：java.lang.Short
最小值：Short.MIN_VALUE=-32768
最大值：Short.MAX_VALUE=32767

基本类型：int 二进制位数：32
包装类：java.lang.Integer
最小值：Integer.MIN_VALUE=-2147483648
最大值：Integer.MAX_VALUE=2147483647

基本类型：long 二进制位数：64
包装类：java.lang.Long
最小值：Long.MIN_VALUE=-9223372036854775808
最大值：Long.MAX_VALUE=9223372036854775807

基本类型：float 二进制位数：32
包装类：java.lang.Float
最小值：Float.MIN_VALUE=1.4E-45
最大值：Float.MAX_VALUE=3.4028235E38

基本类型：double 二进制位数：64
包装类：java.lang.Double
最小值：Double.MIN_VALUE=4.9E-324
最大值：Double.MAX_VALUE=1.7976931348623157E308

基本类型：char 二进制位数：16
包装类：java.lang.Character
最小值：Character.MIN_VALUE=0
最大值：Character.MAX_VALUE=65535
```

Float和Double的最小值和最大值都是以科学记数法的形式输出的，结尾的"E+数字"表示E之前的数字要乘以10的多少倍。比如3.14E3就是3.14×1000=3140，3.14E-3就是3.14/1000=0.00314。

实际上，JAVA中还存在另外一种基本类型void，它也有对应的包装类 java.lang.Void，不过我们无法直接对它们进行操作。

引用类型

- 引用类型变量由类的构造函数创建，可以使用它们访问所引用的对象。这些变量在声明时被指定为一个特定的类型，比如Employee、Pubby等。变量一旦声明后，类型就不能被改变了。
- 对象、数组都是引用数据类型。
- 所有引用类型的默认值都是null。
- 一个引用变量可以用来引用与任何与之兼容的类型。
- 例子：Animal animal = new Animal(“giraffe”)。

Java常量

常量就是一个固定值。它们不需要计算，直接代表相应的值。

常量指不能改变的量。 在Java中用final标志，声明方式和变量类似：

```
final double PI = 3.1415927;
```

虽然常量名也可以用小写，但为了便于识别，通常使用大写字母表示常量。

字面量可以赋给任何内置类型的变量。例如：

```
byte a = 68;
char a = 'A'
```

byte、int、long、和short都可以用十进制、16进制以及8进制的方式来表示。

当使用常量的时候，前缀o表明是8进制，而前缀0x代表16进制。例如：

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

和其他语言一样，Java的字符串常量也是包含在两个引号之间的字符序列。下面是字符串型字面量的例子：

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

字符串常量和字符常量都可以包含任何Unicode字符。例如：

```
char a = '\u0001';
String a = "\u0001";
```

Java语言支持一些特殊的转义字符序列。

符号	字符含义
\n	换行 (0x0a)
\r	回车 (0x0d)
\f	换页符(0x0c)
\b	退格 (0x08)
\s	空格 (0x20)
\t	制表符
\"	双引号
\'	单引号
\	反斜杠
\ddd	八进制字符 (ddd)
\uxxxx	16进制Unicode字符 (xxxx)

这一节讲解了Java的基本数据类型。下一节将探讨不同的变量类型以及它们的用法。

Java变量类型

在Java语言中，所有的变量在使用前必须声明。声明变量的基本格式如下：

```
type identifier [ = value][, identifier [= value] ...] ;
```

格式说明：type为Java数据类型。identifier是变量名。可以使用逗号隔开来声明多个同类型变量。

以下列出了一些变量的声明实例。注意有些包含了初始化过程。

```
int a, b, c;         // 声明三个int型整数：a、 b、c。
int d = 3, e, f = 5; // d声明三个整数并赋予初值。
byte z = 22;         // 声明并初始化z。
double pi = 3.14159; // 声明了pi。
char x = 'x';        // 变量x的值是字符'x'。
```

Java语言支持的变量类型有：

- 局部变量
- 成员变量
- 类变量

Java局部变量

- 局部变量声明在方法、构造方法或者语句块中；
- 局部变量在方法、构造方法、或者语句块被执行的时候创建，当它们执行完成后，变量将会被销毁；
- 访问修饰符不能用于局部变量；
- 局部变量只在声明它的方法、构造方法或者语句块中可见；
- 局部变量是在栈上分配的。
- 局部变量没有默认值，所以局部变量量被声明后，必须经过初始化，才可以使用。

实例1

在以下实例中age是一个局部变量。定义在pubAge()方法中，它的作用域就限制在这个方法中。

```
public class Test{
   public void pupAge(){
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

以上实例编译运行结果如下：

```
Puppy age is: 7
```

实例2

在下面的例子中age变量没有初始化，所以在编译时出错。

```
public class Test{
   public void pupAge(){
      int age;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

以上实例编译运行结果如下:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
         ^
1 error
```

实例变量

- 实例变量声明在一个类中，但在方法、构造方法和语句块之外；
- 当一个对象被实例化之后，每个实例变量的值就跟着确定；
- 实例变量在对象创建的时候创建，在对象被销毁的时候销毁；
- 实例变量的值应该至少被一个方法、构造方法或者语句块引用，使得外部能够通过这些方式获取实例变量信息；
- 实例变量可以声明在使用前或者使用后；
- 访问修饰符可以修饰实例变量；

- 实例变量对于类中的方法、构造方法或者语句块是可见的。一般情况下应该把实例变量设为私有。通过使用访问修饰符可以使实例变量对子类可见；
- 实例变量具有默认值。数值型变量的默认值是0，布尔型变量的默认值是false，引用类型变量的默认值是null。变量的值可以在声明时指定，也可以在构造方法中指定；
- 实例变量可以直接通过变量名访问。但在静态方法以及其他类中，就应该使用完全限定名：ObejectReference.VariableName。

实例：

```
import java.io.*;
public class Employee{
   // 这个成员变量对子类可见
   public String name;
   // 私有变量，仅在该类可见
   private double salary;
   //在构造器中对name赋值
   public Employee (String empName){
      name = empName;
   }
   //设定salary的值
   public void setSalary(double empSal){
      salary = empSal;
   }
   // 打印信息
   public void printEmp(){
      System.out.println("name  : " + name );
      System.out.println("salary :" + salary);
   }

   public static void main(String args[]){
      Employee empOne = new Employee("Ransika");
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

以上实例编译运行结果如下:

```
name  : Ransika
salary :1000.0
```

类变量（静态变量）

- 类变量也称为静态变量，在类中以static关键字声明，但必须在方法构造方法和语句块之外。
- 无论一个类创建了多少个对象，类只拥有类变量的一份拷贝。
- 静态变量除了被声明为常量外很少使用。常量是指声明为publc/private，final和static类型的变量。常量初始化后不可改变。
- 静态变量储存在静态存储区。经常被声明为常量，很少单独使用static声明变量。
- 静态变量在程序开始时创建，在程序结束时销毁。
- 与实例变量具有相似的可见性。但为了对类的使用者可见，大多数静态变量声明为public类型。

- 默认值和实例变量相似。数值型变量默认值是0，布尔型默认值是false，引用类型默认值是null。变量的值可以在声明的时候指定，也可以在构造方法中指定。此外，静态变量还可以在静态语句块中初始化。
- 静态变量可以通过：*ClassName.VariableName*的方式访问。
- 类变量被声明为public static final类型时，类变量名称必须使用大写字母。如果静态变量不是public和final类型，其命名方式与实例变量以及局部变量的命名方式一致。

实例：

```
import java.io.*;
public class Employee{
   //salary是静态的私有变量
   private static double salary;
   // DEPARTMENT是一个常量
   public static final String DEPARTMENT = "Development ";
   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT+"average salary:"+salary);
   }
}
```

以上实例编译运行结果如下:

```
Development average salary:1000
```

注意：如果其他类想要访问该变量，可以这样访问：Employee.DEPARTMENT。

本章节中我们学习了Java的变量类型，下一章节中我们将介绍Java修饰符的使用。

Java修饰符

Java语言提供了很多修饰符，主要分为以下两类：

- 访问修饰符
- 非访问修饰符

修饰符用来定义类、方法或者变量，通常放在语句的最前端。我们通过下面的例子来说明：

```
public class className {
   // ...
}
private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;
public static void main(String[] arguments) {
   // 方法体
}
```

访问控制修饰符

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java支持4种不同的访问权限。

默认的，也称为default，在同一包内可见，不使用任何修饰符。

私有的，以private修饰符指定，在同一类内可见。

共有的，以public修饰符指定，对所有类可见。

受保护的，以protected修饰符指定，对同一包内的类和所有子类可见。

默认访问修饰符-不使用任何关键字

使用默认访问修饰符声明的变量和方法，对同一个包内的类是可见的。接口里的变量都隐式声明为public static final,而接口里的方法默认情况下访问权限为public。

实例：

如下例所示，变量和方法的声明可以不使用任何修饰符。

```
String version = "1.5.1";
boolean processOrder() {
   return true;
}
```

私有访问修饰符-private

私有访问修饰符是最严格的访问级别，所以被声明为private的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为private。

声明为私有访问类型的变量只能通过类中公共的getter方法被外部类访问。

Private访问修饰符的使用主要用来隐藏类的实现细节和保护类的数据。

下面的类使用了私有访问修饰符：

```
public class Logger {
   private String format;
   public String getFormat() {
      return this.format;
   }
   public void setFormat(String format) {
      this.format = format;
   }
}
```

实例中，Logger类中的format变量为私有变量，所以其他类不能直接得到和设置该变量的值。为了使其他类能够操作该变量，定义了两个public方法：getFormat() （返回format的值）和setFormat(String)（设置format的值）

公有访问修饰符-public

被声明为public的类、方法、构造方法和接口能够被任何其他类访问。

如果几个相互访问的public类分布在不用的包中，则需要导入相应public类所在的包。由于类的继承性，类所有的公有方法和变量都能被其子类继承。

以下函数使用了公有访问控制：

```
public static void main(String[] arguments) {
   // ...
}
```

Java程序的main() 方法必须设置成公有的，否则，Java解释器将不能运行该类。

受保护的访问修饰符-protected

被声明为protected的变量、方法和构造器能被同一个包中的任何其他类访问，也能够被不同包中的子类访问。

Protected访问修饰符不能修饰类和接口，方法和成员变量能够声明为protected，但是接口的成员变量和成员方法不能声明为protected。

子类能访问Protected修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。

下面的父类使用了protected访问修饰符，子类重载了父类的openSpeaker()方法。

```
class AudioPlayer {
   protected boolean openSpeaker(Speaker sp) {
      // 实现细节
   }
}

class StreamingAudioPlayer {
   boolean openSpeaker(Speaker sp) {
      // 实现细节
   }
}
```

如果把openSpeaker()方法声明为private，那么除了AudioPlayer之外的类将不能访问该方法。如果把openSpeaker()声明为public，那么所有的类都能够访问该方法。如果我们只想让该方法对其所在类的子类可见，则将该方法声明为protected。

访问控制和继承

请注意以下方法继承的规则：

- 父类中声明为public的方法在子类中也必须为public。
- 父类中声明为protected的方法在子类中要么声明为protected，要么声明为public。不能声明为private。
- 父类中默认修饰符声明的方法，能够在子类中声明为private。
- 父类中声明为private的方法，不能够被继承。

非访问修饰符

为了实现一些其他的功能，Java也提供了许多非访问修饰符。

static修饰符，用来创建类方法和类变量。

Final修饰符，用来修饰类、方法和变量，final修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。

Abstract修饰符，用来创建抽象类和抽象方法。

Synchronized和volatile修饰符，主要用于线程的编程。

Static修饰符

- 静态变量：

  Static关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被成为类变量。局部变量能被声明为static变量。

- 静态方法：

  Static关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

对类变量和方法的访问可以直接使用classname.variablename和classname.methodname的方式访问。

如下例所示，static修饰符用来创建类方法和类变量。

```
public class InstanceCounter {
   private static int numInstances = 0;
   protected static int getCount() {
      return numInstances;
   }

   private static void addInstance() {
      numInstances++;
   }

   InstanceCounter() {
      InstanceCounter.addInstance();
   }

   public static void main(String[] arguments) {
      System.out.println("Starting with " +
      InstanceCounter.getCount() + " instances");
      for (int i = 0; i < 500; ++i){
         new InstanceCounter();
          }
      System.out.println("Created " +
      InstanceCounter.getCount() + " instances");
   }
}
```

以上实例运行编辑结果如下:

```
Started with 0 instances
Created 500 instances
```

Final修饰符

Final变量：

Final变量能被显式地初始化并且只能初始化一次。被声明为final的对象的引用不能指向不同的对象。但是final对象里的数据可以被改变。也就是说final对象的引用不能改变，但是里面的值可以改变。

Final修饰符通常和static修饰符一起使用来创建类常量。

实例:

```
public class Test{
  final int value = 10;
  // 下面是声明常量的实例
  public static final int BOXWIDTH = 6;
  static final String TITLE = "Manager";

  public void changeValue(){
     value = 12; //将输出一个错误
  }
}
```

Final方法

类中的Final方法可以被子类继承，但是不能被子类修改。

声明final方法的主要目的是防止该方法的内容被修改。

如下所示，使用final修饰符声明方法。

```
public class Test{
    public final void changeName(){
       // 方法体
    }
}
```

Final类

Final类不能被继承，没有类能够继承final类的任何特性。

实例：

```
public final class Test {
   // 类体
}
```

Abstract修饰符

抽象类：

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。

一个类不能同时被abstract和final修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。

抽象类可以包含抽象方法和非抽象方法。

实例：

```
abstract class Caravan{
   private double price;
   private String model;
   private String year;
   public abstract void goFast(); //抽象方法
   public abstract void changeColor();
}
```

抽象方法

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。抽象方法不能被声明成final和strict。

任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

抽象方法的声明以分号结尾，例如：public abstract sample();

实例：

```
public abstract class SuperClass{
    abstract void m(); //抽象方法
}

class SubClass extends SuperClass{
     //实现抽象方法
      void m(){
          .........
      }
}
```

Synchronized修饰符

Synchronized关键字声明的方法同一时间只能被一个线程访问。Synchronized修饰符可以应用于四个访问修饰符。

实例：

```
public synchronized void showDetails(){
.......
}
```

Transient修饰符

序列化的对象包含被transient修饰的实例变量时，java虚拟机(JVM)跳过该特定的变量。

该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

实例：

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

volatile修饰符

Volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。一个volatile对象引用可能是null。

实例：

```
public class MyRunnable implements Runnable
{
    private volatile boolean active;
    public void run()
    {
        active = true;
        while (active) // line 1
        {
            // 代码
        }
    }
    public void stop()
    {
        active = false; // line 2
    }
}
```

一般地，在一个线程中调用run()方法，在另一个线程中调用stop()方法。如果line 1中的active位于缓冲区的值被使用，那么当把line 2中的active设置成false时，循环也不会停止。

Java运算符

计算机的最基本用途之一就是执行数学运算，作为一门计算机语言，Java也提供了一套丰富的运算符来操纵变量。我们可以把运算符分成以下几组：

- 算术运算符
- 关系运算符
- 位运算符
- 逻辑运算符
- 赋值运算符
- 其他运算符

算术运算符

算术运算符用在数学表达式中，它们的作用和在数学中的作用一样。下表列出了所有的算术运算符。

表格中的实例假设整数变量A的值为10，变量B的值为20：

操作符	描述	例子
+	加法 - 相加运算符两侧的值	A + B等于30
-	减法 - 左操作数减去右操作数	A – B等于-10
*	乘法 - 相乘操作符两侧的值	A * B等于200
/	除法 - 左操作数除以右操作数	B / A等于2
%	取模 - 右操作数除左操作数的余数	B%A等于0
+ +	自增 - 操作数的值增加1	B + +等于21
-	自减 - 操作数的值减少1	B - -等于19

实例

下面的简单示例程序演示了算术运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     int c = 25;
     int d = 25;
     System.out.println("a + b = " + (a + b) );
     System.out.println("a - b = " + (a - b) );
     System.out.println("a * b = " + (a * b) );
     System.out.println("b / a = " + (b / a) );
     System.out.println("b % a = " + (b % a) );
     System.out.println("c % a = " + (c % a) );
     System.out.println("a++   = " +  (a++) );
     System.out.println("b--   = " +  (a--) );
     // Check the difference in d++ and ++d
     System.out.println("d++   = " +  (d++) );
     System.out.println("++d   = " +  (++d) );
  }
}
```

以上实例编译运行结果如下：

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

关系运算符

下表为Java支持的关系运算符

表格中的实例整数变量A的值为10，变量B的值为20：

运算符	描述	例子
==	检查如果两个操作数的值是否相等，如果相等则条件为真。	（A == B）为假(非真)。
=!	检查如果两个操作数的值是否相等，如果值不相等则条件为真。	（A =! B）为真。
>	检查左操作数的值是否大于右操作数的值，如果是那么条件为真。	（A > B）非真。
<	检查左操作数的值是否小于右操作数的值，如果是那么条件为真。	（A < B）为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真。	（A >= B）为假。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真。	（A <= B）为真。

实例

下面的简单示例程序演示了关系运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     System.out.println("a == b = " + (a == b) );
     System.out.println("a != b = " + (a != b) );
     System.out.println("a > b = " + (a > b) );
     System.out.println("a < b = " + (a < b) );
     System.out.println("b >= a = " + (b >= a) );
     System.out.println("b <= a = " + (b <= a) );
  }
}
```

以上实例编译运行结果如下：

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

位运算符

Java定义了位运算符，应用于整数类型(int)，长整型(long)，短整型(short)，字符型(char)，和字节型(byte)等类型。

位运算符作用在所有的位上，并且按位运算。假设a = 60，和b = 13;它们的二进制格式表示将如下：

```
A = 0011 1100
B = 0000 1101
-----------------
A&b = 0000 1100
A | B = 0011 1101
^ B = 0011 0001
~A= 1100 0011
```

下表列出了位运算符的基本运算,假设整数变量A的值为60和变量B的值为13：

操作符	描述	例子
&	按位与操作符，当且仅当两个操作数的某一位都非0时候结果的该位才为1。	（A&B），得到12，即0000 1100
|	按位或操作符，只要两个操作数的某一位有一个非0时候结果的该位就为1。	（A | B）得到61，即 0011 1101
^	按位异或操作符，两个操作数的某一位不相同时候结果的该位就为1。	（A ^ B）得到49，即 0011 0001
?	按位补运算符翻转操作数的每一位。	（?A）得到-60，即1100 0011
<<	按位左移运算符。左操作数按位左移右操作数指定的位数。	A << 2得到240，即 1111 0000
>>	按位右移运算符。左操作数按位右移右操作数指定的位数。	A >> 2得到15即 1111
>>>	按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充。	A>>>2得到15即0000 1111

实例

下面的简单示例程序演示了位运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {
  public static void main(String args[]) {
     int a = 60; /* 60 = 0011 1100 */
     int b = 13; /* 13 = 0000 1101 */
     int c = 0;
     c = a & b;       /* 12 = 0000 1100 */
     System.out.println("a & b = " + c );

     c = a | b;       /* 61 = 0011 1101 */
     System.out.println("a | b = " + c );

     c = a ^ b;       /* 49 = 0011 0001 */
     System.out.println("a ^ b = " + c );

     c = ~a;          /*-61 = 1100 0011 */
     System.out.println("~a = " + c );

     c = a << 2;     /* 240 = 1111 0000 */
     System.out.println("a << 2 = " + c );

     c = a >> 2;     /* 215 = 1111 */
     System.out.println("a >> 2  = " + c );

     c = a >>> 2;     /* 215 = 0000 1111 */
     System.out.println("a >>> 2 = " + c );
  }
}
```

以上实例编译运行结果如下：

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

逻辑运算符

下表列出了逻辑运算符的基本运算，假设布尔变量A为真，变量B为假

操作符	描述	例子
&&	称为逻辑与运算符。当且仅当两个操作数都为真，条件才为真。	（A && B）为假。
||	称为逻辑或操作符。如果任何两个操作数任何一个为真，条件为真。	（A || B）为真。
!	称为逻辑非运算符。用来反转操作数的逻辑状态。如果条件为true，则逻辑非运算符将得到false。	！（A && B）为真。

实例

下面的简单示例程序演示了逻辑运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {
  public static void main(String args[]) {
     boolean a = true;
     boolean b = false;
     System.out.println("a && b = " + (a&&b));
     System.out.println("a || b = " + (a||b) );
     System.out.println("!(a && b) = " + !(a && b));
  }
}
```

以上实例编译运行结果如下：

```
a && b = false
a || b = true
!(a && b) = true
```

赋值运算符

下面是Java语言支持的赋值运算符：

操作符	描述	例子
=	简单的赋值运算符，将右操作数的值赋给左侧操作数	C = A + B将把A + B得到的值赋给C
+=	加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数	C += A等价于C = C + A
-=	减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数	C -= A等价于C = C - A
*=	乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数	C = *A*等价于*C* = *C* A
/=	除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数	C /= A等价于C = C / A
%=	取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数	C %= A等价于C = C%A
<<=	左移位赋值运算符	C < = 2等价于C = C << 2
>>=	右移位赋值运算符	C >>= 2等价于C = C >> 2
&=	按位与赋值运算符	C &= 2等价于C = C&2
^=	按位异或赋值操作符	C ^= 2等价于C = C ^ 2
|=	按位或赋值操作符	C |= 2等价于C = C | 2

实例

面的简单示例程序演示了赋值运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {
  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     int c = 0;
     c = a + b;
     System.out.println("c = a + b = " + c );
     c += a ;
     System.out.println("c += a  = " + c );
     c -= a ;
     System.out.println("c -= a = " + c );
     c *= a ;
     System.out.println("c *= a = " + c );
     a = 10;
     c = 15;
     c /= a ;
     System.out.println("c /= a = " + c );
     a = 10;
     c = 15;
     c %= a ;
     System.out.println("c %= a  = " + c );
     c <<= 2 ;
     System.out.println("c <<= 2 = " + c );
     c >>= 2 ;
     System.out.println("c >>= 2 = " + c );
     c >>= 2 ;
     System.out.println("c >>= a = " + c );
     c &= a ;
     System.out.println("c &= 2  = " + c );
     c ^= a ;
     System.out.println("c ^= a   = " + c );
     c |= a ;
     System.out.println("c |= a   = " + c );
  }
}
```

以上实例编译运行结果如下：

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

条件运算符（?:）

条件运算符也被称为三元运算符。该运算符有3个操作数，并且需要判断布尔表达式的值。该运算符的主要是决定哪个值应该赋值给变量。

```
variable x = (expression) ? value if true : value if false
```

实例

```
public class Test {
   public static void main(String args[]){
      int a , b;
      a = 10;
b = (a == 1) ? 20: 30;
System.out.println( "Value of b is : " +  b );
      b = (a == 10) ? 20: 30;
     System.out.println( "Value of b is : " + b );
   }
}
```

以上实例编译运行结果如下：

```
Value of b is : 30
Value of b is : 20
```

instanceOf 运算符

该运算符用于操作对象实例，检查该对象是否是一个特定类型（类类型或接口类型）。

instanceof运算符使用格式如下：

```
( Object reference variable ) instanceOf  (class/interface type)
```

如果运算符左侧变量所指的对象，是操作符右侧类或接口(class/interface)的一个对象，那么结果为真。

下面是一个例子：

```
String name = 'James';
boolean result = name instanceOf String; // 由于name是Strine类型，所以返回真
```

如果被比较的对象兼容于右侧类型,该运算符仍然返回true。

看下面的例子：

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result);
   }
}
```

以上实例编译运行结果如下：

```
true
```

Java运算符优先级

当多个运算符出现在一个表达式中，谁先谁后呢？这就涉及到运算符的优先级别的问题。在一个多运算符的表达式中，运算符优先级不同会导致最后得出的结果差别甚大。

例如，（1+3）＋（3+2）*2，这个表达式如果按加号最优先计算，答案就是 18，如果按照乘号最优先，答案则是 14。

再如，x = 7 + 3 *2;这里x得到13，而不是20，因为乘法运算符比加法运算符有较高的优先级，所以先计算3* 2得到6，然后再加7。

下表中具有最高优先级的运算符在的表的最上面，最低优先级的在表的底部。

类别	操作符	关联性		
后缀	() [] . (点操作符)	左到右		
一元	+ + - ！?	从右到左		
乘性	* /%	左到右		
加性	+ -	左到右		
移位	>> >>> <<	左到右		
关系	>> = << =	左到右		
相等	== !=	左到右		
按位与	＆	左到右		
按位异或	^	左到右		
按位或			左到右	
逻辑与	&&	左到右		
逻辑或				左到右
条件	？：	从右到左		
赋值	= + = - = * = / =%= >> = << =&= ^ =	=	从右到左	
逗号	，	左到右		

Java循环结构 - for, while 及 do...while

顺序结构的程序语句只能被执行一次。如果您想要同样的操作执行多次,，就需要使用循环结构。

Java中有三种主要的循环结构：

- while循环
- do…while循环
- for循环

在Java5中引入了一种主要用于数组的增强型for循环。

while循环

while是最基本的循环，它的结构为：

```
while( 布尔表达式 ) {
    //循环内容
}
```

只要布尔表达式为true，循环体会一直执行下去。

实例

```
public class Test {
   public static void main(String args[]) {
      int x = 10;
      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

do…while循环

对于while语句而言，如果不满足条件，则不能进入循环。但有时候我们需要即使不满足条件，也至少执行一次。

do…while循环和while循环相似，不同的是，do…while循环至少会执行一次。

```
do {
       //代码语句
}while(布尔表达式);
```

注意：布尔表达式在循环体的后面，所以语句块在检测布尔表达式之前已经执行了。 如果布尔表达式的值为true，则语句块一直执行，直到布尔表达式的值为false。

实例

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      do{
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }while( x < 20 );
   }
}
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

for循环

虽然所有循环结构都可以用while或者do...while表示，但Java提供了另一种语句 —— for循环，使一些循环结构变得更加简单。

for循环执行的次数是在执行前就确定的。语法格式如下

```
for(初始化; 布尔表达式; 更新) {
    //代码语句
}
```

关于for循环有以下几点说明：

- 最先执行初始化步骤。可以声明并初始化一个或多个循环控制变量，也可以是空语句。
- 然后，检测布尔表达式的值。如果为true，循环体被执行。如果为false，循环终止，开始执行循环体后面的语句。
- 执行一次循环后，更新循环控制变量。
- 再次检测布尔表达式。循环执行上面的过程。

实例

```
public class Test {

   public static void main(String args[]) {

      for(int x = 10; x < 20; x = x+1) {
         System.out.print("value of x : " + x );
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Java增强for循环

Java5引入了一种主要用于数组的增强型for循环。

Java增强for循环语法格式如下:

```
for(声明语句 : 表达式)
{
   //代码句子
}
```

声明语句：声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。

表达式：表达式是要访问的数组名，或者是返回值为数组的方法。

实例

```
public class Test {

   public static void main(String args[]){
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names ={"James", "Larry", "Tom", "Lacy"};
      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

以上实例编译运行结果如下：

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

break关键字

break主要用在循环语句或者switch语句中，用来跳出整个语句块。

break跳出最里层的循环，并且继续执行该循环下面的语句。

语法

break的用法很简单，就是循环结构中的一条语句：

```
break;
```

实例

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
          break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
10
20
```

continue关键字

continue适用于任何循环控制结构中。作用是让程序立刻跳转到下一次循环的迭代。

在for循环中，continue语句使程序立即跳转到更新语句。

在while或者do…while循环中，程序立即跳转到布尔表达式的判断语句。

语法

continue就是循环体中一条简单的语句：

```
continue;
```

实例

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
          continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
10
20
40
50
```

Java分支结构 - if...else/switch

顺序结构只能顺序执行，不能进行判断和选择，因此需要分支结构。

Java有两种分支结构：

- if语句
- switch语句

if语句

一个if语句包含一个布尔表达式和一条或多条语句。

语法

If语句的用语法如下：

```
if(布尔表达式)
{
   //如果布尔表达式为true将执行的语句
}
```

如果布尔表达式的值为true，则执行if语句中的代码块。否则执行If语句块后面的代码。

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      if( x < 20 ){
         System.out.print("这是 if 语句");
      }
   }
}
```

以上代码编译运行结果如下：

```
这是 if 语句
```

if...else语句

if语句后面可以跟else语句，当if语句的布尔表达式值为false时，else语句块会被执行。

语法

if…else的用法如下：

```
if(布尔表达式){
   //如果布尔表达式的值为true
}else{
   //如果布尔表达式的值为false
}
```

实例

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x < 20 ){
         System.out.print("这是 if 语句");
      }else{
         System.out.print("这是 else 语句");
      }
   }
}
```

以上代码编译运行结果如下：

```
这是 else 语句
```

if...else if...else语句

if语句后面可以跟elseif…else语句，这种语句可以检测到多种可能的情况。

使用if，else if，else语句的时候，需要注意下面几点：

- if语句至多有1个else语句，else语句在所有的elseif语句之后。
- If语句可以有若干个elseif语句，它们必须在else语句之前。
- 一旦其中一个else if语句检测为true，其他的else if以及else语句都将跳过执行。

语法

if...else语法格式如下:

```
if(布尔表达式 1){
   //如果布尔表达式 1的值为true执行代码
}else if(布尔表达式 2){
   //如果布尔表达式 2的值为true执行代码
}else if(布尔表达式 3){
   //如果布尔表达式 3的值为true执行代码
}else {
   //如果以上布尔表达式都不为true执行代码
}
```

实例

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x == 10 ){
         System.out.print("Value of X is 10");
      }else if( x == 20 ){
         System.out.print("Value of X is 20");
      }else if( x == 30 ){
         System.out.print("Value of X is 30");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

以上代码编译运行结果如下：

```
Value of X is 30
```

嵌套的if…else语句

使用嵌套的if-else语句是合法的。也就是说你可以在另一个if或者elseif语句中使用if或者elseif语句。

语法

嵌套的if…else语法格式如下：

```
if(布尔表达式 1){
   ////如果布尔表达式 1的值为true执行代码
   if(布尔表达式 2){
      ////如果布尔表达式 2的值为true执行代码
   }
}
```

你可以像 *if* 语句一样嵌套 *else if...else*。

实例

```
public class Test {

   public static void main(String args[]){
      int x = 30;
      int y = 10;

      if( x == 30 ){
         if( y == 10 ){
             System.out.print("X = 30 and Y = 10");
          }
       }
    }
}
```

以上代码编译运行结果如下：

```
X = 30 and Y = 10
```

switch语句

switch语句判断一个变量与一系列值中某个值是否相等，每个值称为一个分支。

语法

switch语法格式如下：

```
switch(expression){
    case value :
       //语句
       break; //可选
    case value :
       //语句
       break; //可选
    //你可以有任意数量的case语句
    default : //可选
       //语句
}
```

switch语句有如下规则：

- switch语句中的变量类型只能为byte、short、int或者char。
- switch语句可以拥有多个case语句。每个case后面跟一个要比较的值和冒号。
- case语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量。
- 当变量的值与case语句的值相等时，那么case语句之后的语句开始执行，直到break语句出现才会跳出switch语句。3
- 当遇到break语句时，switch语句终止。程序跳转到switch语句后面的语句执行。case语句不必须要包含break语句。如果没有break语句出现，程序会继续执行下一条case语句，直到出现break语句。

- switch语句可以包含一个default分支，该分支必须是switch语句的最后一个分支。default在没有case语句的值和变量值相等的时候执行。default分支不需要break语句。

实例

```
public class Test {

   public static void main(String args[]){
      //char grade = args[0].charAt(0);
      char grade = 'C';

      switch(grade)
      {
         case 'A' :
            System.out.println("Excellent!");
            break;
         case 'B' :
         case 'C' :
            System.out.println("Well done");
            break;
         case 'D' :
            System.out.println("You passed");
         case 'F' :
            System.out.println("Better try again");
            break;
         default :
            System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

以上代码编译运行结果如下：

```
$ java Test
Well done
Your grade is a C
$
```

Java Number类

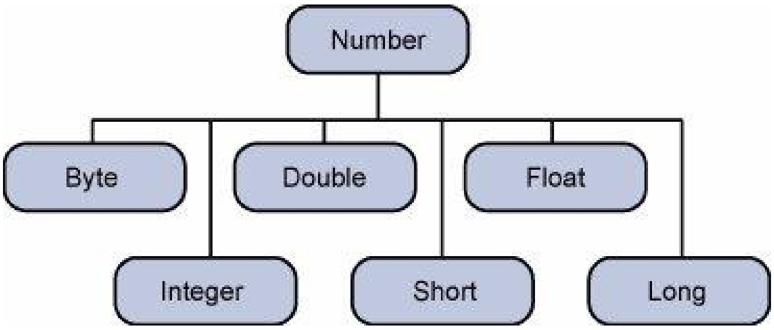
一般地，当需要使用数字的时候，我们通常使用内置数据类型，如：byte、int、long、double 等。

实例

```
int i = 5000;
float gpa = 13.65;
byte mask = 0xaf;
```

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情形。为了解决这个问题，Java语言为每一个内置数据类型提供了对应的包装类。

所有的包装类（Integer、Long、Byte、Double、Float、Short）都是抽象类Number的子类。



这种由编译器特别支持的包装称为装箱，所以当内置数据类型被当作对象使用的时候，编译器会把内置类型装箱为包装类。相似的，编译器也可以把一个对象拆箱为内置类型。Number类属于java.lang包。

下面是一个装箱与拆箱的例子：

```
public class Test{

   public static void main(String args[]){
      Integer x = 5; // boxes int to an Integer object
      x =  x + 10;   // unboxes the Integer to a int
      System.out.println(x);
   }
}
```

以上实例编译运行结果如下：

```
15
```

当x被赋为整型值时，由于x是一个对象，所以编译器要对x进行装箱。然后，为了使x能进行加运算，所以要对x进行拆箱。

Number类的成员方法

下面的表中列出的是Number类的方法：

方法	描述
xxxValue()	将number对象转换为xxx数据类型的值并返回。
compareTo()	将number对象与参数比较。
equals()	判断number对象是否与参数相等。
valueOf()	返回一个Integer对象指定的内置数据类型
toString()	以字符串形式返回值。
parseInt()	将字符串解析为int类型。
abs()	返回参数的绝对值。
ceil()	对整形变量向左取整，返回类型为double型。
floor()	对整型变量向右取整。返回类型为double类型。
rint()	返回与参数最接近的整数。返回类型为double。
round()	返回一个最接近的int、long型值。
min()	返回两个参数中的最小值。
max()	返回两个参数中的最大值。
exp()	返回自然数底数e的参数次方。
log()	返回参数的自然数底数的对数值。
pow()	返回第一个参数的第二个参数次方。
sqrt()	求参数的算术平方根。
sin()	求指定double类型参数的正弦值。
cos()	求指定double类型参数的余弦值。
tan()	求指定double类型参数的正切值。
asin()	求指定double类型参数的反正弦值。
acos()	求指定double类型参数的反余弦值。
atan()	求指定double类型参数的反正切值。
atan2()	将笛卡尔坐标转换为极坐标，并返回极坐标的角度值。
toDegrees()	将参数转化为角度。
toRadians()	将角度转换为弧度。
random()	返回一个随机数。

Java Character类

使用字符时，我们通常使用的是内置数据类型char。

实例

```
char ch = 'a';

// Unicode for uppercase Greek omega character
char uniChar = '\u039A';

// 字符数组
char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };
```

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情况。为了解决这个问题，Java语言为内置数据类型char提供了包装类Character类。

Character类提供了一系列方法来操纵字符。你可以使用Character的构造方法创建一个Character类对象，例如：

```
Character ch = new Character('a');
```

在某些情况下，Java编译器会自动创建一个Character对象。

例如，将一个char类型的参数传递给需要一个Character类型参数的方法时，那么编译器会自动地将char类型参数转换为Character对象。 这种特征称为装箱，反过来称为拆箱。

实例

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch = 'a';

// Here primitive 'x' is boxed for method test,
// return is unboxed to char 'c'
char c = test('x');
```

转义序列

前面有反斜杠（\）的字符代表转义字符，它对编译器来说是有特殊含义的。

下面列表展示了Java的转义序列：

转义序列	描述
\t	在文中该处插入一个tab键
\b	在文中该处插入一个后退键
\n	在文中该处换行
\r	在文中该处插入回车
\f	在文中该处插入换页符
\'	在文中该处插入单引号
\"	在文中该处插入双引号
\	在文中该处插入反斜杠

实例

当打印语句遇到一个转义序列时，编译器可以正确地对其进行解释。

```
public class Test {

   public static void main(String args[]) {
      System.out.println("She said \"Hello!\" to me.");
   }
}
```

以上实例编译运行结果如下：

```
She said "Hello!" to me.
```

Character 方法

下面是Character类的方法：

方法	描述
isLetter()	是否是一个字母
isDigit()	是否是一个数字字符
isWhitespace()	是否一个空格
isUpperCase()	是否是大写字母
isLowerCase()	是否是小写字母
toUpperCase()	指定字母的大写形式
toLowerCase()	指定字母的小写形式
toString()	返回字符的字符串形式，字符串的长度仅为1

对于方法的完整列表，请参考的java.lang.Character API规范。

Java String类

字符串广泛应用在Java编程中，在Java中字符串属于对象，Java提供了String类来创建和操作字符串。

创建字符串

创建字符串最简单的方式如下:

```
String greeting = "Hello world!";
```

在代码中遇到字符串常量时，这里的值是"Hello world!"，编译器会使用该值创建一个String对象。

和其它对象一样，可以使用关键字和构造方法来创建String对象。

String类有11种构造方法，这些方法提供不同的参数来初始化字符串，比如提供一个字符数组参数:

```
public class StringDemo{

   public static void main(String args[]){
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

以上实例编译运行结果如下：

```
hello.
```

注意:String类是不可改变的，所以你一旦创建了String对象，那它的值就无法改变了。 如果需要对字符串做很多修改，那么应该选择使用StringBuffer & StringBuilder 类。

字符串长度

用于获取有关对象的信息的方法称为访问器方法。

String类的一个访问器方法是length()方法，它返回字符串对象包含的字符数。

下面的代码执行后，len变量等于17:

```
public class StringDemo {

   public static void main(String args[]) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println( "String Length is : " + len );
   }
}
```

以上实例编译运行结果如下：

```
String Length is : 17
```

连接字符串

String类提供了连接两个字符串的方法：

```
string1.concat(string2);
```

返回string2连接string1的新字符串。也可以对字符串常量使用concat()方法，如：

```
"My name is ".concat("Zara");
```

更常用的是使用'+'操作符来连接字符串，如：

```
"Hello," + " world" + "!"
```

结果如下:

```
"Hello, world!"
```

下面是一个例子:

```
public class StringDemo {
   public static void main(String args[]) {
   String string1 = "saw I was ";
   System.out.println("Dot " + string1 + "Tod");
}
}
```

以上实例编译运行结果如下：

```
Dot saw I was Tod
```

创建格式化字符串

我们知道输出格式化数字可以使用printf()和format()方法。String类使用静态方法format()返回一个String对象而不是PrintStream对象。

String类的静态方法format()能用来创建可复用的格式化字符串，而不仅仅是用于一次打印输出。如下所示：

```
System.out.printf("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

你也可以这样写

```
String fs;
fs = String.format("The value of the float variable is " +
                   "%f, while the value of the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String 方法

下面是String类支持的方法，更多详细，参看Java API文档:

方法	描述
char charAt(int index)	返回指定索引处的 char 值。
int compareTo(Object o)	把这个字符串和另一个对象比较。
int compareTo(String anotherString)	按字典顺序比较两个字符串。
int compareToIgnoreCase(String str)	按字典顺序比较两个字符串，不考虑大小写。
String concat(String str)	将指定字符串连接到此字符串的结尾。
boolean contentEquals(StringBuffer sb)	当且仅当字符串与指定的StringButter有相同顺序的字符时候返回真。
static String copyValueOf(char[] data)	返回指定数组中表示该字符序列的 String。
static String copyValueOf(char[] data, int offset, int count)	返回指定数组中表示该字符序列的 String。
boolean endsWith(String suffix)	测试此字符串是否以指定的后缀结束。
boolean equals(Object anObject)	将此字符串与指定的对象比较。
boolean equalsIgnoreCase(String anotherString)	将此 String 与另一个 String 比较，不考虑大小写。

byte[] getBytes()	使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
byte[] getBytes(String charsetName)	使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	将字符从此字符串复制到目标字符数组。
int hashCode()	返回此字符串的哈希码。
int indexOf(int ch)	返回指定字符在此字符串中第一次出现处的索引。
int indexOf(int ch, int fromIndex)	返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
int indexOf(String str)	返回指定子字符串在此字符串中第一次出现处的索引。
int indexOf(String str, int fromIndex)	返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。
String intern()	返回字符串对象的规范化表示形式。
int lastIndexOf(int ch)	返回指定字符在此字符串中最后一次出现处的索引。
int lastIndexOf(int ch, int fromIndex)	返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。
int lastIndexOf(String str)	返回指定子字符串在此字符串中最右边出现处的索引。
int lastIndexOf(String str, int fromIndex)	返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。
int length()	返回此字符串的长度。
boolean matches(String regex)	告知此字符串是否匹配给定的正则表达式。
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等。
boolean regionMatches(int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等。
String replace(char oldChar, char newChar)	返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。
String replaceAll(String regex, String replacement)	使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
String replaceFirst(String regex, String	使用给定的 replacement 替换此字符串匹配

replacement)	给定的正则表达式的第一个子字符串。
String[] split(String regex)	根据给定正则表达式的匹配拆分此字符串。
String[] split(String regex, int limit)	根据匹配给定的正则表达式来拆分此字符串。
boolean startsWith(String prefix)	测试此字符串是否以指定的前缀开始。
boolean startsWith(String prefix, int toffset)	测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
CharSequence subSequence(int beginIndex, int endIndex)	返回一个新的字符序列，它是此序列的一个子序列。
String substring(int beginIndex)	返回一个新的字符串，它是此字符串的一个子字符串。
String substring(int beginIndex, int endIndex)	返回一个新字符串，它是此字符串的一个子字符串。
char[] toCharArray()	将此字符串转换为一个新的字符数组。
String toLowerCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为小写。
String toLowerCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。
String toString()	返回此对象本身（它已经是一个字符串！）。
String toUpperCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为大写。
String toUpperCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。
String trim()	返回字符串的副本，忽略前导空白和尾部空白。
static String valueOf(primitive data type x)	返回给定data type类型x参数的字符串表示形式。

Java StringBuffer和StringBuilder类

当对字符串进行修改的时候，需要使用StringBuffer和StringBuilder类。

和String类不同的是，StringBuffer和StringBuilder类的对象能够被多次的修改，并且不产生新的未使用对象。

StringBuilder类在Java 5中被提出，它和StringBuffer之间的最大不同在于StringBuilder的方法不是线程安全的（不能同步访问）。

由于StringBuilder相较于StringBuffer有速度优势，所以多数情况下建议使用StringBuilder类。然而在应用程序要求线程安全的情况下，则必须使用StringBuffer类。

实例

```
public class Test{

    public static void main(String args[]){
       StringBuffer sBuffer = new StringBuffer(" test");
       sBuffer.append(" String Buffer");
       System.ou.println(sBuffer);
   }
}
```

以上实例编译运行结果如下：

```
test String Buffer
```

StringBuffer 方法

以下是StringBuffer类支持的主要方法：

方法	描述
public StringBuffer append(String s)	将指定的字符串追加到此字符序列。
public StringBuffer reverse()	将此字符序列用其反转形式取代。
public delete(int start, int end)	移除此序列的子字符串中的字符。
public insert(int offset, int i)	将 `int` 参数的字符串表示形式插入此序列中。
replace(int start, int end, String str)	使用给定 `String` 中的字符替换此序列的子字符串中的字符。

下面的列表里的方法和String类的方法类似：

方法	描述
int capacity()	返回当前容量。
char charAt(int index)	返回此序列中指定索引处的 `char` 值。
void ensureCapacity(int minimumCapacity)	确保容量至少等于指定的最小值。
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	将字符从此序列复制到目标字符数组 `dst` 。
int indexOf(String str)	返回第一次出现的指定子字符串在该字符串中的索引。
int indexOf(String str, int fromIndex)	从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。
int lastIndexOf(String str)	返回最右边出现的指定子字符串在此字符串中的索引。
int lastIndexOf(String str, int fromIndex)	返回最后一次出现的指定子字符串在此字符串中的索引。
int length()	返回长度（字符数）。
void setCharAt(int index, char ch)	将给定索引处的字符设置为 `ch` 。
void setLength(int newLength)	设置字符序列的长度。
CharSequence subSequence(int start, int end)	返回一个新的字符序列，该字符序列是此序列的子序列。
String substring(int start)	返回一个新的 `String`，它包含此字符序列当前所包含的字符子序列。
String substring(int start, int end)	返回一个新的 `String` ，它包含此序列当前所包含的字符子序列。
String toString()	返回此序列中数据的字符串表示形式。

Java 数组

数组对于每一门编辑应语言来说都是重要的数据结构之一，当然不同语言对数组的实现及处理也不尽相同。

Java语言中提供的数组是用来存储固定大小的同类型元素。

你可以声明一个数组变量，如numbers[100]来代替直接声明100个独立变量number0，number1，....，number99。

本教程将为大家介绍Java数组的声明、创建和初始化，并给出其对应的代码。

声明数组变量

首先必须声明数组变量，才能在程序中使用数组。下面是声明数组变量的语法：

```
dataType[] arrayRefVar;   // 首选的方法

或

dataType arrayRefVar[];  // 效果相同，但不是首选方法
```

注意: 建议使用dataType[] arrayRefVar 的声明风格声明数组变量。 dataType arrayRefVar[] 风格是来自 C/C++ 语言 ，在Java中采用是为了让 C/C++ 程序员能够快速理解java语言。

实例

下面是这两种语法的代码示例：

```
double[] myList;         // 首选的方法

或

double myList[];         //  效果相同，但不是首选方法
```

创建数组

Java语言使用new操作符来创建数组，语法如下：

```
arrayRefVar = new dataType[arraySize];
```

上面的语法语句做了两件事：

- 一、使用dataType[arraySize]创建了一个数组。
- 二、把新创建的数组的引用赋值给变量 arrayRefVar。

数组变量的声明，和创建数组可以用一条语句完成，如下所示：

```
dataType[] arrayRefVar = new dataType[arraySize];
```

另外，你还可以使用如下的方式创建数组。

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

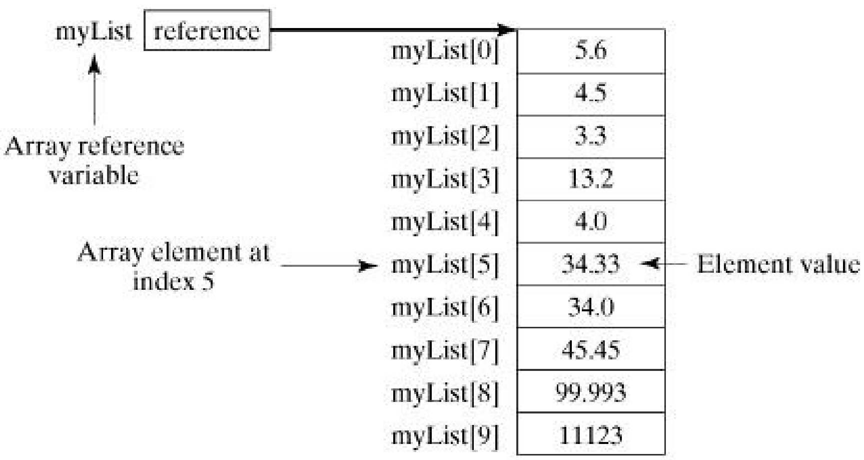
数组的元素是通过索引访问的。数组索引从0开始，所以索引值从0到arrayRefVar.length-1。

实例

下面的语句首先声明了一个数组变量myList，接着创建了一个包含10个double类型元素的数组，并且把它的引用赋值给myList变量。

```
double[] myList = new double[10];
```

下面的图片描绘了数组myList。这里myList数组里有10个double元素，它的下标从0到9。



处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本循环或者foreach循环。

示例

该实例完整地展示了如何创建、初始化和操纵数组：

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // 打印所有数组元素
      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");
      }
      // 计算所有元素的总和
      double total = 0;
      for (int i = 0; i < myList.length; i++) {
         total += myList[i];
      }
      System.out.println("Total is " + total);
      // 查找最大元素
      double max = myList[0];
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) max = myList[i];
      }
      System.out.println("Max is " + max);
   }
}
```

以上实例编译运行结果如下：

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

foreach循环

JDK 1.5 引进了一种新的循环类型，被称为foreach循环或者加强型循环，它能在不使用下标的情况下遍历数组。

示例

该实例用来显示数组myList中的所有元素：

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // 打印所有数组元素
      for (double element: myList) {
         System.out.println(element);
      }
   }
}
```

以上实例编译运行结果如下：

```
1.9
2.9
3.4
3.5
```

数组作为函数的参数

数组可以作为参数传递给方法。例如，下面的例子就是一个打印int数组中元素的方法。

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

下面例子调用printArray方法打印出 3，1，2，6，4和2：

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

数组作为函数的返回值

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }
  return result;
}
```

以上实例中result数组作为函数的返回值。

Arrays 类

java.util.Arrays类能方便地操作数组，它提供的所有方法都是静态的。具有以下功能：

- 给数组赋值：通过fill方法。
- 对数组排序：通过sort方法,按升序。
- 比较数组：通过equals方法比较数组中元素值是否相等。
- 查找数组元素：通过binarySearch方法能对排序好的数组进行二分查找法操作。

具体说明请查看下表：

方法	说明
public static int binarySearch(Object[] a, Object key)	用二分查找算法在给定数组中搜索给定值的对象(Byte,Int,double等)。数组在调用前必须排序好的。如果查找值包含在数组中，则返回搜索键的索引；否则返回 (-(插入点) - 1)。
public static boolean equals(long[] a, long[] a2)	如果两个指定的 long 型数组彼此相等，则返回 true。如果两个数组包含相同数量的元素，并且两个数组中的所有相应元素对都是相等的，则认为这两个数组是相等的。换句话说，如果两个数组以相同顺序包含相同的元素，则两个数组是相等的。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。
public static void fill(int[] a, int val)	将指定的 int 值分配给指定 int 型数组指定范围中的每个元素。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。
public static void sort(Object[] a)	对指定对象数组根据其元素的自然顺序进行升序排列。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。

Java 日期时间

java.util包提供了Date类来封装当前的日期和时间。 Date类提供两个构造函数来实例化Date对象。

第一个构造函数使用当前日期和时间来初始化对象。

```
Date( )
```

第二个构造函数接收一个参数，该参数是从1970年1月1日起的微秒数。

```
Date(long millisec)
```

Date对象创建以后，可以调用下面的方法。

方法	描述
boolean after(Date date)	若当调用此方法的Date对象在指定日期之后返回true,否则返回false。
boolean before(Date date)	若当调用此方法的Date对象在指定日期之前返回true,否则返回false。
Object clone( )	返回此对象的副本。
int compareTo(Date date)	比较当调用此方法的Date对象和指定日期。两者相等时候返回0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数。
int compareTo(Object obj)	若obj是Date类型则操作等同于compareTo(Date) 。否则它抛出ClassCastException。
boolean equals(Object date)	当调用此方法的Date对象和指定日期相等时候返回true,否则返回false。
long getTime( )	返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
int hashCode( )	返回此对象的哈希码值。
void setTime(long time)	用自1970年1月1日00:00:00 GMT以后time毫秒数设置时间和日期。
String toString( )	转换Date对象为String表示形式，并返回该字符串。

获取当前日期时间

Java中获取当前日期和时间很简单，使用Date对象的 toString()方法来打印当前日期和时间，如下所示：

```
import java.util.Date;

public class DateDemo {
   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 使用 toString() 函数显示日期时间
       System.out.println(date.toString());
   }
}
```

以上实例编译运行结果如下:

```
Mon May 04 09:51:52 CDT 2013
```

日期比较

Java使用以下三种方法来比较两个日期：

- 使用getTime( ) 方法获取两个日期（自1970年1月1日经历的微妙数值），然后比较这两个值。
- 使用方法before()，after()和equals()。例如，一个月的12号比18号早，则new Date(99, 2, 12).before(new Date (99, 2, 18))返回true。
- 使用compareTo()方法，它是由Comparable接口定义的，Date类实现了这个接口。

使用**SimpleDateFormat**格式化日期

SimpleDateFormat是一个以语言环境敏感的方式来格式化和分析日期的类。SimpleDateFormat允许你选择任何用户自定义日期时间格式来运行。例如：

```
import java.util.*;
import java.text.*;

public class DateDemo {
   public static void main(String args[]) {

      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

      System.out.println("Current Date: " + ft.format(dNow));
   }
}
```

以上实例编译运行结果如下:

```
Current Date: Sun 2004.07.18 at 04:14:09 PM PDT
```

简单的DateFormat格式化编码

时间模式字符串用来指定时间格式。在此模式中，所有的ASCII字母被保留为模式字母，定义如下：

字母	描述	示例
G	纪元标记	AD
y	四位年份	2001
M	月份	July or 07
d	一个月的日期	10
h	A.M./P.M. (1~12)格式小时	12
H	一天中的小时 (0~23)	22
m	分钟数	30
s	秒数	55
S	微妙数	234
E	星期几	Tuesday
D	一年中的日子	360
F	一个月中第几周的周几	2 (second Wed. in July)
w	一年中第几周	40
W	一个月中第几周	1
a	A.M./P.M. 标记	PM
k	一天中的小时(1~24)	24
K	A.M./P.M. (0~11)格式小时	10
z	时区	Eastern Standard Time
'	文字定界符	Delimiter
"	单引号	`

使用printf格式化日期

printf方法可以很轻松地格式化时间和日期。使用两个字母格式，它以t开头并且以下面表格中的一个字母结尾。例如：

```
import java.util.Date;

public class DateDemo {

  public static void main(String args[]) {
     // 初始化 Date 对象
     Date date = new Date();

     // 使用toString()显示日期和时间
     String str = String.format("Current Date/Time : %tc", date );

     System.out.printf(str);
  }
}
```

以上实例编译运行结果如下:

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

如果你需要重复提供日期，那么利用这种方式来格式化它的每一部分就有点复杂了。因此，可以利用一个格式化字符串指出要被格式化的参数的索引。

索引必须紧跟在%后面，而且必须以$结束。例如：

```
import java.util.Date;

public class DateDemo {

   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 使用toString()显示日期和时间
       System.out.printf("%1$s %2$tB %2$td, %2$tY",
                         "Due date:", date);
   }
}
```

以上实例编译运行结果如下:

```
Due date: February 09, 2004
```

或者，你可以使用<标志。它表明先前被格式化的参数要被再次使用。例如：

```
import java.util.Date;

public class DateDemo {

   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 显示格式化时间
       System.out.printf("%s %tB %<te, %<tY",
                         "Due date:", date);
   }
}
```

以上实例编译运行结果如下:

```
Due date: February 09, 2004
```

日期和时间转换字符

字符	描述	例子
c	完整的日期和时间	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 格式日期	2004-02-09
D	U.S. 格式日期 (月/日/年)	02/09/2004
T	24小时时间	18:05:19
r	12小时时间	06:05:19 pm
R	24小时时间，不包含秒	18:05
Y	4位年份(包含前导0)	2004
y	年份后2位(包含前导0)	04
C	年份前2位(包含前导0)	20
B	月份全称	February
b	月份简称	Feb
n	2位月份(包含前导0)	02
d	2位日子(包含前导0)	03
e	2位日子(不包含前导0)	9
A	星期全称	Monday
a	星期简称	Mon
j	3位年份(包含前导0)	069
H	2位小时(包含前导0), 00 到 23	18
k	2位小时(不包含前导0), 0 到 23	18
I	2位小时(包含前导0), 01 到 12	06
l	2位小时(不包含前导0), 1 到 12	6
M	2位分钟(包含前导0)	05
S	2位秒数(包含前导0)	19
L	3位毫秒(包含前导0)	047
N	9位纳秒(包含前导0)	047000000

P	大写上下午标志	PM
p	小写上下午标志	pm
z	从GMT的RFC 822数字偏移	-0800
Z	时区	PST
s	自 1970-01-01 00:00:00 GMT的秒数	1078884319
Q	自 1970-01-01 00:00:00 GMT的毫妙	1078884319047

还有其他有用的日期和时间相关的类。对于更多的细节，你可以参考到Java标准文档。

解析字符串为时间

SimpleDateFormat 类有一些附加的方法，特别是parse()，它试图按照给定的SimpleDateFormat 对象的格式化存储来解析字符串。例如：

```
import java.util.*;
import java.text.*;

public class DateDemo {

   public static void main(String args[]) {
      SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

      String input = args.length == 0 ? "1818-11-11" : args[0];

      System.out.print(input + " Parses as ");

      Date t;

      try {
          t = ft.parse(input);
          System.out.println(t);
      } catch (ParseException e) {
          System.out.println("Unparseable using " + ft);
      }
   }
}
```

以上实例编译运行结果如下:

```
$ java DateDemo
1818-11-11 Parses as Wed Nov 11 00:00:00 GMT 1818
$ java DateDemo 2007-12-01
2007-12-01 Parses as Sat Dec 01 00:00:00 GMT 2007
```

Java 休眠(sleep)

你可以让程序休眠一毫秒的时间或者到您的计算机的寿命长的任意段时间。例如，下面的程序会休眠10秒：

```
import java.util.*;

public class SleepDemo {
   public static void main(String args[]) {
      try {
         System.out.println(new Date( ) + "\n");
         Thread.sleep(5*60*10);
         System.out.println(new Date( ) + "\n");
      } catch (Exception e) {
          System.out.println("Got an exception!");
      }
   }
}
```

以上实例编译运行结果如下:

```
Sun May 03 18:04:41 GMT 2009

Sun May 03 18:04:51 GMT 2009
```

测量时间

下面的一个例子表明如何测量时间间隔（以毫秒为单位）：

```
import java.util.*;

public class DiffDemo {

   public static void main(String args[]) {
      try {
         long start = System.currentTimeMillis( );
         System.out.println(new Date( ) + "\n");
         Thread.sleep(5*60*10);
         System.out.println(new Date( ) + "\n");
         long end = System.currentTimeMillis( );
         long diff = end - start;
         System.out.println("Difference is : " + diff);
      } catch (Exception e) {
         System.out.println("Got an exception!");
      }
   }
}
```

以上实例编译运行结果如下:

```
Sun May 03 18:16:51 GMT 2009

Sun May 03 18:16:57 GMT 2009

Difference is : 5993
```

Calendar类

我们现在已经能够格式化并创建一个日期对象了，但是我们如何才能设置和获取日期数据的特定部分呢，比如说小时，日，或者分钟? 我们又如何在日期的这些部分加上或者减去值呢? 答案是使用Calendar 类。

Calendar类的功能要比Date类强大很多，而且在实现方式上也比Date类要复杂一些。

Calendar类是一个抽象类，在实际使用时实现特定的子类的对象，创建对象的过程对程序员来说是透明的，只需要使用getInstance方法创建即可。

创建一个代表系统当前日期的Calendar对象

```
Calendar c = Calendar.getInstance();//默认是当前日期
```

创建一个指定日期的Calendar对象

使用Calendar类代表特定的时间，需要首先创建一个Calendar的对象，然后再设定该对象中的年月日参数来完成。

```
//创建一个代表2009年6月12日的Calendar对象
Calendar c1 = Calendar.getInstance();
c1.set(2009, 6 - 1, 12);
```

Calendar类对象字段类型

Calendar类中用一下这些常量表示不同的意义，jdk内的很多类其实都是采用的这种思想

常量	描述
Calendar.YEAR	年份
Calendar.MONTH	月份
Calendar.DATE	日期
Calendar.DAY_OF_MONTH	日期，和上面的字段意义完全相同
Calendar.HOUR	12小时制的小时
Calendar.HOUR_OF_DAY	24小时制的小时
Calendar.MINUTE	分钟
Calendar.SECOND	秒
Calendar.DAY_OF_WEEK	星期几

Calendar类对象信息的设置

Set设置

如：

```
Calendar c1 = Calendar.getInstance();
```

调用：

```
public final void set(int year,int month,int date)
```

```
c1.set(2009, 6 - 1, 12);//把Calendar对象c1的年月日分别设这为：2009、6、12
```

利用字段类型设置

如果只设定某个字段，例如日期的值，则可以使用如下set方法：

```
public void set(int field,int value)
```

把 c1对象代表的日期设置为10号，其它所有的数值会被重新计算

```
c1.set(Calendar.DATE,10);
```

把c1对象代表的年份设置为2008年，其他的所有数值会被重新计算

```
c1.set(Calendar.YEAR,2008);
```

其他字段属性set的意义以此类推

Add设置

```
Calendar c1 = Calendar.getInstance();
```

把c1对象的日期加上10，也就是c1所表的日期的10天后的日期，其它所有的数值会被重新计算

```
c1.add(Calendar.DATE, 10);
```

把c1对象的日期加上10，也就是c1所表的日期的10天前的日期，其它所有的数值会被重新计算

```
<pre>c1.add(Calendar.DATE, -10);
```

其他字段属性的add的意义以此类推

Calendar类对象信息的获得

```
Calendar c1 = Calendar.getInstance();
// 获得年份
int year = c1.get(Calendar.YEAR);
// 获得月份
int month = c1.get(Calendar.MONTH) + 1;
// 获得日期
int date = c1.get(Calendar.DATE);
// 获得小时
int hour = c1.get(Calendar.HOUR_OF_DAY);
// 获得分钟
int minute = c1.get(Calendar.MINUTE);
// 获得秒
int second = c1.get(Calendar.SECOND);
// 获得星期几（注意（这个与Date类是不同的）：1代表星期日、2代表星期1、3代表星期二，以此类推）
int day = c1.get(Calendar.DAY_OF_WEEK);
```

GregorianCalendar类

Calendar类实现了公历日历，GregorianCalendar是Calendar类的一个具体实现。

Calendar 的getInstance（）方法返回一个默认用当前的语言环境和时区初始化的GregorianCalendar对象。GregorianCalendar定义了两个字段：AD和BC。这些代表公历定义的两个时代。

下面列出GregorianCalendar对象的几个构造方法：

构造函数	说明
GregorianCalendar()	在具有默认语言环境的默认时区内使用当前时间构造一个默认的 GregorianCalendar。
GregorianCalendar(int year, int month, int date)	在具有默认语言环境的默认时区内构造一个带有给定日期设置的 GregorianCalendar
GregorianCalendar(int year, int month, int date, int hour, int minute)	为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。
GregorianCalendar(int year, int month, int date, int hour, int minute, int second)	为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。
GregorianCalendar(Locale aLocale)	在具有给定语言环境的默认时区内构造一个基于当前时间的 GregorianCalendar。
GregorianCalendar(TimeZone zone)	在具有默认语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。
GregorianCalendar(TimeZone zone, Locale aLocale)	在具有给定语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。

这里是GregorianCalendar 类提供的一些有用的方法列表：

方法	说明
void add(int field, int amount)	根据日历规则，将指定的（有符号的）时间量添加到给定的日历字段中。
protected void computeFields()	转换UTC毫秒值为时间域值
protected void computeTime()	覆盖Calendar，转换时间域值为UTC毫秒值
boolean equals(Object obj)	比较此 GregorianCalendar 与指定的 Object。
int get(int field)	获取指定字段的时间值
int getActualMaximum(int field)	返回当前日期，给定字段的最大值
int getActualMinimum(int field)	返回当前日期，给定字段的最小值
int getGreatestMinimum(int field)	返回此 GregorianCalendar 实例给定日历字段的最高的最小值。
Date getGregorianChange()	获得格里高利历的更改日期。
int getLeastMaximum(int field)	返回此 GregorianCalendar 实例给定日历字段的最低的最大值
int getMaximum(int field)	返回此 GregorianCalendar 实例的给定日历字段的最大值。
Date getTime()	获取日历当前时间。
long getTimeInMillis()	获取用长整型表示的日历的当前时间
TimeZone getTimeZone()	获取时区。
int getMinimum(int field)	返回给定字段的最小值。
int hashCode()	重写hashCode.
boolean isLeapYear(int year)	确定给定的年份是否为闰年。
void roll(int field, boolean up)	在给定的时间字段上添加或减去（上/下）单个时间单元，不更改更大的字段。
void set(int field, int value)	用给定的值设置时间字段。
void set(int year, int month, int date)	设置年、月、日的值。
void set(int year, int month, int date, int hour, int minute)	设置年、月、日、小时、分钟的值。
void set(int year, int month, int date, int hour, int minute, int second)	设置年、月、日、小时、分钟、秒的值。
void setGregorianChange(Date date)	设置 GregorianCalendar 的更改日期。
void setTime(Date date)	用给定的日期设置Calendar的当前时间。

void setTimeInMillis(long millis)	用给定的long型毫秒数设置Calendar的当前时间。
void setTimeZone(TimeZone value)	用给定时区值设置当前时区。
String toString()	返回代表日历的字符串。

实例

```
import java.util.*;

public class GregorianCalendarDemo {

   public static void main(String args[]) {
      String months[] = {
      "Jan", "Feb", "Mar", "Apr",
      "May", "Jun", "Jul", "Aug",
      "Sep", "Oct", "Nov", "Dec"};

      int year;
      // 初始化 Gregorian 日历
      // 使用当前时间和日期
      // 默认为本地时间和时区
      GregorianCalendar gcalendar = new GregorianCalendar();
      // 显示当前时间和日期的信息
      System.out.print("Date: ");
      System.out.print(months[gcalendar.get(Calendar.MONTH)]);
      System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
      System.out.println(year = gcalendar.get(Calendar.YEAR));
      System.out.print("Time: ");
      System.out.print(gcalendar.get(Calendar.HOUR) + ":");
      System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
      System.out.println(gcalendar.get(Calendar.SECOND));

      // 测试当前年份是否为闰年
      if(gcalendar.isLeapYear(year)) {
         System.out.println("当前年份是闰年");
      }
      else {
         System.out.println("当前年份不是闰年");
      }
   }
}
```

以上实例编译运行结果如下：

```
Date: Apr 22 2009
Time: 11:25:27
当前年份不是闰年
```

关于Calender 类的完整列表，你可以参考标准的Java文档。

Java正则表达式

正则表达式定义了字符串的模式。

正则表达式可以用来搜索、编辑或处理文本。

正则表达式并不仅限于某一种语言，但是在每种语言中有细微的差别。

Java正则表达式和Perl的是最为相似的。

java.util.regex包主要包括以下三个类：

- **Pattern**类：

  pattern对象是一个正则表达式的编译表示。Pattern类没有公共构造方法。要创建一个Pattern对象，你必须首先调用其公共静态编译方法，它返回一个Pattern对象。该方法接受一个正则表达式作为它的第一个参数。

- **Matcher**类：

  Matcher对象是对输入字符串进行解释和匹配操作的引擎。与Pattern类一样，Matcher也没有公共构造方法。你需要调用Pattern对象的matcher方法来获得一个Matcher对象。

- **PatternSyntaxException**：

  PatternSyntaxException是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

捕获组

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

例如，正则表达式(dog) 创建了单一分组，组里包含"d"，"o"，和"g"。

捕获组是通过从左至右计算其开括号来编号。例如，在表达式（（A）（B（C））），有四个这样的组：

- ((A)(B(C)))
- (A)
- (B(C))
- (C)

可以通过调用matcher对象的groupCount方法来查看表达式有多少个分组。groupCount方法返回一个int值，表示matcher对象当前有多个捕获组。

还有一个特殊的组（组0），它总是代表整个表达式。该组不包括在groupCount的返回值中。

实例

下面的例子说明如何从一个给定的字符串中找到数字串：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    public static void main( String args[] ){

      // 按指定模式在字符串查找
      String line = "This order was placed for QT3000! OK?";
      String pattern = "(.*)(\\d+)(.*)";

      // 创建 Pattern 对象
      Pattern r = Pattern.compile(pattern);

      // 现在创建 matcher 对象
      Matcher m = r.matcher(line);
      if (m.find( )) {
         System.out.println("Found value: " + m.group(0) );
         System.out.println("Found value: " + m.group(1) );
         System.out.println("Found value: " + m.group(2) );
      } else {
         System.out.println("NO MATCH");
      }
   }
}
```

以上实例编译运行结果如下：

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0
```

正则表达式语法

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。序列"\"匹配"\"，"("匹配"("。
^	匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与"\n"或"\r"之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与"\n"或"\r"之前的位置匹配。
*	零次或多次匹配前面的字符或子表达式。例如，zo 匹配*"z"和"zoo"*。 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，

+	但与"z"不匹配。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o*"。
{*n*,*m*}	*M* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（*、+、?、{*n*}、{*n*,}、{*n*,*m*}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。
.	匹配除"\n"之外的任何单个字符。若要匹配包括"\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。
(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"("或者")"。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。
(?=*pattern*)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
(?!*pattern*)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的"Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x|*y*	匹配 *x* 或 *y*。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。
[*xyz*]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^*xyz*]	反向字符集。匹配未包含的任何字符。例如，"abc"匹配"plain"中的"p"。
[*a-z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a-z*]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。

\b	的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\c*x*	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。
\D	非数字字符匹配。等效于 0-9。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效。
\W	与任何非单词字符匹配。与"[^A-Za-z0-9_]"等效。
\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。
num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。
_n_	标识一个八进制转义码或反向引用。如果 _n 前面至少有 _*n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。
_nm_	标识一个八进制转义码或反向引用。如果 _nm 前面至少有 _*nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 _nm 前面至少有 _*n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 _nm 匹配八进制值 _*nm*，其中 *n* 和 *m* 是八进制数字 (0-7)。
\nml	当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。
\u*n*	匹配 *n*，其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (?)。

Mather类的方法

索引方法

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

方法	说明
public int start()	返回以前匹配的初始索引。
public int start(int group)	返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引
public int end()	返回最后匹配字符之后的偏移量。
public int end(int group)	返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

研究方法

研究方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

方法	说明	
public boolean lookingAt()	尝试将从区域开头开始的输入序列与该模式匹配。	
public boolean find()	尝试查找与该模式匹配的输入序列的下一个子序列。	
public boolean find(int start	)	重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。
public boolean matches()	尝试将整个区域与模式匹配。	

替换方法

替换方法是替换输入字符串里文本的方法：

方法	说明
public Matcher appendReplacement(StringBuffer sb, String replacement)	实现非终端添加和替换步骤。
public StringBuffer appendTail(StringBuffer sb)	实现终端添加和替换步骤。
public String replaceAll(String replacement)	替换模式与给定替换字符串相匹配的输入序列的每个子序列。
public String replaceFirst(String replacement)	替换模式与给定替换字符串匹配的输入序列的第一个子序列。
public static String quoteReplacement(String s)	返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给Matcher类的appendReplacement 方法一个字面字符串一样工作。

start 和end 方法

下面是一个对单词"cat"出现在输入字符串中出现次数进行计数的例子：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT =
                                    "cat cat cat cattie cat";

    public static void main( String args[] ){
       Pattern p = Pattern.compile(REGEX);
       Matcher m = p.matcher(INPUT); // 获取 matcher 对象
       int count = 0;

       while(m.find()) {
         count++;
         System.out.println("Match number "+count);
         System.out.println("start(): "+m.start());
         System.out.println("end(): "+m.end());
      }
   }
}
```

以上实例编译运行结果如下：

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

可以看到这个例子是使用单词边界，以确保字母 "c" "a" "t" 并非仅是一个较长的词的子串。它也提供了一些关于输入字符串中匹配发生位置的有用信息。

Start方法返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引，end方法最后一个匹配字符的索引加1。

matches 和lookingAt 方法

matches 和lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是matcher要求整个序列都匹配，而lookingAt 不要求。

这两个方法经常在输入字符串的开始使用。

我们通过下面这个例子，来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main( String args[] ){
       pattern = Pattern.compile(REGEX);
       matcher = pattern.matcher(INPUT);

       System.out.println("Current REGEX is: "+REGEX);
       System.out.println("Current INPUT is: "+INPUT);

       System.out.println("lookingAt(): "+matcher.lookingAt());
       System.out.println("matches(): "+matcher.matches());
   }
}
```

以上实例编译运行结果如下：

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

replaceFirst 和replaceAll 方法

replaceFirst 和replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

下面的例子来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. " +
                                    "All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
       Pattern p = Pattern.compile(REGEX);
       // get a matcher object
       Matcher m = p.matcher(INPUT);
       INPUT = m.replaceAll(REPLACE);
       System.out.println(INPUT);
   }
}
```

以上实例编译运行结果如下：

```
The cat says meow. All cats say meow.
```

appendReplacement 和 appendTail 方法

Matcher 类也提供了appendReplacement 和appendTail 方法用于文本替换：

看下面的例子来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
   private static String REGEX = "a*b";
   private static String INPUT = "aabfooaabfooabfoob";
   private static String REPLACE = "-";
   public static void main(String[] args) {
      Pattern p = Pattern.compile(REGEX);
      // 获取 matcher 对象
      Matcher m = p.matcher(INPUT);
      StringBuffer sb = new StringBuffer();
      while(m.find()){
         m.appendReplacement(sb,REPLACE);
      }
      m.appendTail(sb);
      System.out.println(sb.toString());
   }
}
```

以上实例编译运行结果如下：

```
-foo-foo-foo-
```

PatternSyntaxException 类的方法

PatternSyntaxException 是一个非强制异常类，它指示一个正则表达式模式中的语法错误。

PatternSyntaxException 类提供了下面的方法来帮助我们查看发生了什么错误。

方法	说明
public String getDescription()	获取错误的描述。
public int getIndex()	获取错误的索引。
public String getPattern()	获取错误的正则表达式模式。
public String getMessage()	返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。

Java 方法

在前面几个章节中我们经常使用到System.out.println()，那么它是什么呢？

println()是一个方法(Method)，而System是系统类(Class)，out是标准输出对象(Object)。这句话的用法是调用系统类System中的标准输出对象out中的方法println()。

那么什么是方法呢？

Java方法是语句的集合，它们在一起执行一个功能。

- 方法是解决一类问题的步骤的有序组合
- 方法包含于类或对象中
- 方法在程序中被创建，在其他地方被引用

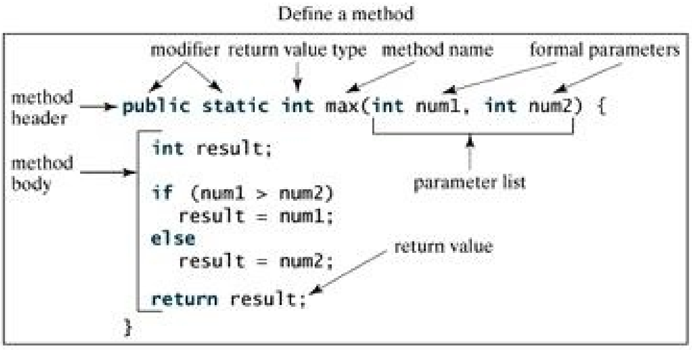
方法的定义

一般情况下，定义一个方法包含以下语法：

```
修饰符 返回值类型 方法名 (参数类型 参数名){
    ...
    方法体
    ...
    return 返回值;
}
```

方法包含一个方法头和一个方法体。下面是一个方法的所有部分：

- 修饰符：修饰符，这是可选的，告诉编译器如何调用该方法。定义了该方法的访问类型。
- 返回值类型 ：方法可能会返回值。returnValueType是方法返回值的数据类型。有些方法执行所需的操作，但没有返回值。在这种情况下，returnValueType是关键字**void**。
- 方法名：是方法的实际名称。方法名和参数表共同构成方法签名。
- 参数类型：参数像是一个占位符。当方法被调用时，传递值给参数。这个值被称为实参或变量。参数列表是指方法的参数类型、顺序和参数的个数。参数是可选的，方法可以不包含任何参数。
- 方法体：方法体包含具体的语句，定义该方法的功能。



如：

```
public static int age(int birthday){...}
```

参数可以有多个：

```
static float interest(float principal, int year){...}
```

注意： 在一些其它语言中方法指过程和函数。一个返回非void类型返回值的方法称为函数；一个返回void类型返回值的方法叫做过程。

实例

下面的方法包含2个参数num1和num2，它返回这两个参数的最大值。

```
/** 返回两个整型变量数据的较大值 */
public static int max(int num1, int num2) {
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

方法调用

Java支持两种调用方法的方式，根据方法是否返回值来选择。

当程序调用一个方法时，程序的控制权交给了被调用的方法。当被调用方法的返回语句执行或者到达方法体闭括号时候交还控制权给程序。

当方法返回一个值的时候，方法调用通常被当做一个值。例如：

```
int larger = max(30, 40);
```

如果方法返回值是void，方法调用一定是一条语句。例如，方法println返回void。下面的调用是个语句：

```
System.out.println("Welcome to Java!");
```

示例

下面的例子演示了如何定义一个方法，以及如何调用它：

```
public class TestMax {
   /** 主方法 */
   public static void main(String[] args) {
      int i = 5;
      int j = 2;
      int k = max(i, j);
      System.out.println("The maximum between " + i +
                    " and " + j + " is " + k);
   }

   /** 返回两个整数变量较大的值 */
   public static int max(int num1, int num2) {
      int result;
      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
}
```

以上实例编译运行结果如下：

```
The maximum between 5 and 2 is 5
```

这个程序包含main方法和max方法。Main方法是被JVM调用的，除此之外，main方法和其它方法没什么区别。

main方法的头部是不变的，如例子所示，带修饰符public和static,返回void类型值，方法名字是main,此外带个一个String[]类型参数。String[]表明参数是字符串数组。

void 关键字

本节说明如何声明和调用一个void方法。

下面的例子声明了一个名为printGrade的方法，并且调用它来打印给定的分数。

示例

```
public class TestVoidMethod {

   public static void main(String[] args) {
      printGrade(78.5);
   }

   public static void printGrade(double score) {
      if (score >= 90.0) {
         System.out.println('A');
      }
      else if (score >= 80.0) {
         System.out.println('B');
      }
      else if (score >= 70.0) {
         System.out.println('C');
      }
      else if (score >= 60.0) {
         System.out.println('D');
      }
      else {
         System.out.println('F');
      }
   }
}
```

以上实例编译运行结果如下：

```
C
```

这里printGrade方法是一个void类型方法，它不返回值。

一个void方法的调用一定是一个语句。 所以，它被在main方法第三行以语句形式调用。就像任何以分号结束的语句一样。

通过值传递参数

调用一个方法时候需要提供参数，你必须按照参数列表指定的顺序提供。

例如，下面的方法连续n次打印一个消息：

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

示例

下面的例子演示按值传递的效果。

该程序创建一个方法，该方法用于交换两个变量。

```
public class TestPassByValue {

   public static void main(String[] args) {
      int num1 = 1;
      int num2 = 2;

      System.out.println("Before swap method, num1 is " +
                          num1 + " and num2 is " + num2);

      // 调用swap方法
      swap(num1, num2);
      System.out.println("After swap method, num1 is " +
                         num1 + " and num2 is " + num2);
   }
   /** 交换两个变量的方法 */
   public static void swap(int n1, int n2) {
      System.out.println("\tInside the swap method");
      System.out.println("\t\tBefore swapping n1 is " + n1
                           + " n2 is " + n2);
      // 交换 n1 与 n2的值
      int temp = n1;
      n1 = n2;
      n2 = temp;

      System.out.println("\t\tAfter swapping n1 is " + n1
                           + " n2 is " + n2);
   }
}
```

以上实例编译运行结果如下：

```
Before swap method, num1 is 1 and num2 is 2
        Inside the swap method
                Before swapping n1 is 1 n2 is 2
                After swapping n1 is 2 n2 is 1
After swap method, num1 is 1 and num2 is 2
```

传递两个参数调用swap方法。有趣的是，方法被调用后，实参的值并没有改变。

方法的重载

上面使用的max方法仅仅适用于int型数据。但如果你想得到两个浮点类型数据的最大值呢？

解决方法是创建另一个有相同名字但参数不同的方法，如下面代码所示：

```
public static double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

如果你调用max方法时传递的是int型参数，则 int型参数的max方法就会被调用；

如果传递的事double型参数，则double类型的max方法体会被调用，这叫做方法重载；

就是说一个类的两个方法拥有相同的名字，但是有不同的参数列表。

Java编译器根据方法签名判断哪个方法应该被调用。

方法重载可以让程序更清晰易读。执行密切相关任务的方法应该使用相同的名字。

重载的方法必须拥有不同的参数列表。你不能仅仅依据修饰符或者返回类型的不同来重载方法。

变量作用域

变量的范围是程序中该变量可以被引用的部分。

方法内定义的变量被称为局部变量。

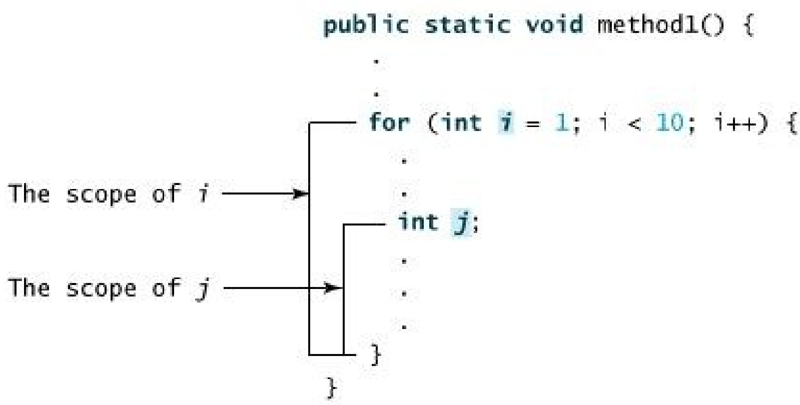
局部变量的作用范围从声明开始，直到包含它的块结束。

局部变量必须声明才可以使用。

方法的参数范围涵盖整个方法。参数实际上是一个局部变量。

for循环的初始化部分声明的变量，其作用范围在整个循环。

但循环体内声明的变量其适用范围是从它声明到循环体结束。它包含如下所示的变量声明：



你可以在一个方法里，不同的非嵌套块中多次声明一个具有相同的名称局部变量，但你不能在嵌套块内两次声明局部变量。

命令行参数的使用

有时候你希望运行一个程序时候再传递给它消息。这要靠传递命令行参数给main()函数实现。

命令行参数是在执行程序时候紧跟在程序名字后面的信息。

实例

下面的程序打印所有的命令行参数：

```
public class CommandLine {

   public static void main(String args[]){
      for(int i=0; i<args.length; i++){
         System.out.println("args[" + i + "]: " +
                                          args[i]);
      }
   }
}
```

如下所示，运行这个程序：

```
java CommandLine this is a command line 200 -100
```

运行结果如下：

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

构造方法

当一个对象被创建时候，构造方法用来初始化该对象。构造方法和它所在类的名字相同，但构造方法没有返回值。

通常会使用构造方法给一个类的实例变量赋初值，或者执行其它必要的步骤来创建一个完整的对象。

不管你与否自定义构造方法，所有的类都有构造方法，因为Java自动提供了一个默认构造方法，它把所有成员初始化为0。

一旦你定义了自己的构造方法，默认构造方法就会失效。

实例

下面是一个使用构造方法的例子：

```
// 一个简单的构造函数
class MyClass {
   int x;

   // 以下是构造函数
   MyClass() {
      x = 10;
   }
}
```

你可以像下面这样调用构造方法来初始化一个对象：

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.x + " " + t2.x);
   }
}
```

大多时候需要一个有参数的构造方法。

实例

下面是一个使用构造方法的例子：

```
// 一个简单的构造函数
class MyClass {
   int x;

   // 以下是构造函数
   MyClass(int i ) {
      x = i;
   }
}
```

你可以像下面这样调用构造方法来初始化一个对象：

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

运行结果如下：

```
10 20
```

可变参数

JDK 1.5 开始，Java支持传递同类型的可变参数给一个方法。

方法的可变参数的声明如下所示：

```
typeName... parameterName
```

在方法声明中，在指定参数类型后加一个省略号(...) 。

一个方法中只能指定一个可变参数，它必须是方法的最后一个参数。任何普通的参数必须在它之前声明。

实例

```
public class VarargsDemo {

   public static void main(String args[]) {
      // 调用可变参数的方法
      printMax(34, 3, 3, 2, 56.5);
      printMax(new double[]{1, 2, 3});
   }

   public static void printMax( double... numbers) {
   if (numbers.length == 0) {
      System.out.println("No argument passed");
      return;
   }

   double result = numbers[0];

   for (int i = 1; i <  numbers.length; i++)
      if (numbers[i] >  result)
      result = numbers[i];
      System.out.println("The max value is " + result);
   }
}
```

以上实例编译运行结果如下：

```
The max value is 56.5
The max value is 3.0
```

finalize() 方法

Java允许定义这样的方法，它在对象被垃圾收集器析构(回收)之前调用，这个方法叫做finalize( )，它用来清除回收对象。

例如，你可以使用finalize()来确保一个对象打开的文件被关闭了。

在finalize()方法里，你必须指定在对象销毁时候要执行的操作。

finalize()一般格式是：

```
protected void finalize()
{
   // 在这里终结代码
}
```

关键字protected是一个限定符，它确保finalize() 方法不会被该类以外的代码调用。

当然，Java的内存回收可以由JVM来自动完成。如果你手动使用，则可以使用上面的方法。

实例

```
public class FinalizationDemo {
    public static void main(String[] args) {
        Cake c1 = new Cake(1);
        Cake c2 = new Cake(2);
        Cake c3 = new Cake(3);

        c2 = c3 = null;
        System.gc(); //调用Java垃圾收集器
    }
}

class Cake extends Object {
    private int id;
    public Cake(int id) {
        this.id = id;
        System.out.println("Cake Object " + id + "is created");
    }

    protected void finalize() throws java.lang.Throwable {
        super.finalize();
        System.out.println("Cake Object " + id + "is disposed");
    }
}
```

运行以上代码，输出结果如下：

```
C:\1>java FinalizationDemo
Cake Object 1is created
Cake Object 2is created
Cake Object 3is created
Cake Object 3is disposed
Cake Object 2is disposed
```

Java 流(Stream)、文件(File)和IO

Java.io包几乎包含了所有操作输入、输出需要的类。所有这些流类代表了输入源和输出目标。

Java.io包中的流支持很多种格式，比如：基本类型、对象、本地化字符集等等。

一个流可以理解为一个数据的序列。输入流表示从一个源读取数据，输出流表示向一个目标写数据。

Java为I/O提供了强大的而灵活的支持，使其更广泛地应用到文件传输和网络编程中。

但本节讲述最基本的和流与I/O相关的功能。我们将通过一个个例子来学习这些功能。

读取控制台输入

Java的控制台输入由Sysem.in完成。

为了获得一个绑定到控制台的字符流，你可以把System.in包装在一个BufferedReader 对象中来创建一个字符流。

下面是创建BufferedReader的基本语法：

```
BufferedReader br = new BufferedReader(new
                      InputStreamReader(System.in));
```

BufferedReader对象创建后，我们便可以使用read()方法从控制台读取一个字符，或者用readLine()方法读取一个字符串。

从控制台读取多字符输入

从BufferedReader对象读取一个字符要使用read()方法，它的语法如下：

```
int read( ) throws IOException
```

每次调用read()方法，它从输入流读取一个字符并把该字符作为整数值返回。 当流结束的时候返回-1。该方法抛出IOException。

下面的程序示范了用read()方法从控制台不断读取字符直到用户输入"q"。

```
// 使用 BufferedReader 在控制台读取字符

import java.io.*;

public class BRRead {
   public static void main(String args[]) throws IOException
   {
      char c;
      // 使用 System.in 创建 BufferedReader
      BufferedReader br = new BufferedReader(new
                         InputStreamReader(System.in));
      System.out.println("Enter characters, 'q' to quit.");
      // 读取字符
      do {
         c = (char) br.read();
         System.out.println(c);
      } while(c != 'q');
   }
}
```

以上实例编译运行结果如下:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

从控制台读取字符串

从标准输入读取一个字符串需要使用BufferedReader的readLine()方法。

它的一般格式是：

```
String readLine( ) throws IOException
```

下面的程序读取和显示字符行直到你输入了单词"end"。

```
// 使用 BufferedReader 在控制台读取字符
import java.io.*;
public class BRReadLines {
   public static void main(String args[]) throws IOException
   {
      // 使用 System.in 创建 BufferedReader
      BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));
      String str;
      System.out.println("Enter lines of text.");
      System.out.println("Enter 'end' to quit.");
      do {
         str = br.readLine();
         System.out.println(str);
      } while(!str.equals("end"));
   }
}
```

以上实例编译运行结果如下:

```
Enter lines of text.
Enter 'end' to quit.
This is line one
This is line one
This is line two
This is line two
end
end
```

控制台输出

在此前已经介绍过，控制台的输出由 print( ) 和println( )完成。这些方法都由类PrintStream 定义，System.out是该类对象的一个引用。

PrintStream 继承了OutputStream类，并且实现了方法write()。这样，write()也可以用来往控制台写操作。

PrintStream 定义write()的最简单格式如下所示：

```
void write(int byteval)
```

该方法将byteval的低八位字节写到流中。

实例

下面的例子用write()把字符"A"和紧跟着的换行符输出到屏幕：

```
import java.io.*;

// 演示 System.out.write().
public class WriteDemo {
   public static void main(String args[]) {
      int b;
      b = 'A';
      System.out.write(b);
      System.out.write('\n');
   }
}
```

运行以上实例在输出窗口输出"A"字符

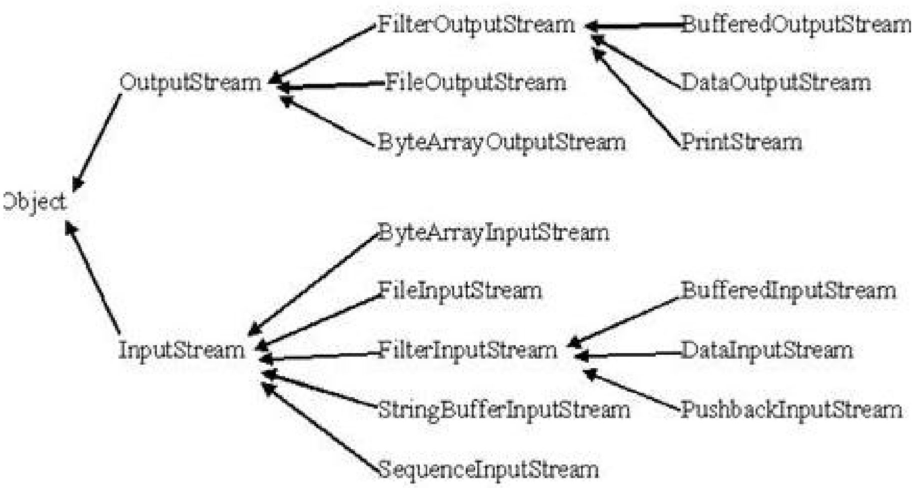
```
A
```

注意：write()方法不经常使用，因为print()和println()方法用起来更为方便。

读写文件

如前所述，一个流被定义为一个数据序列。输入流用于从源读取数据，输出流用于向目标写数据。

下图是一个描述输入流和输出流的类层次图。



下面将要讨论的两个重要的流是FileInputStream 和FileOutputStream：

FileInputStream

该流用于从文件读取数据，它的对象可以用关键字new来创建。

有多种构造方法可用来创建对象。

可以使用字符串类型的文件名来创建一个输入流对象来读取文件：

```
InputStream f = new FileInputStream("C:/java/hello");
```

也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用File()方法来创建一个文件对象：

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

创建了InputStream对象，就可以使用下面的方法来读取流或者进行其他的流操作。

方法	描述
public void close() throws IOException{}	关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。
protected void finalize()throws IOException {}	这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。
public int read(int r)throws IOException{}	这个方法从InputStream对象读取指定字节的数据。返回为整数值。返回下一字节数据，如果已经到结尾则返回-1。
public int read(byte[] r) throws IOException{}	这个方法从输入流读取r.length长度的字节。返回读取的字节数。如果是文件结尾则返回-1。
public int available() throws IOException{}	返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取的字节数。返回一个整数值。

除了InputStream外，还有一些其他的输入流，更多的细节参考下面链接：

- ByteArrayInputStream
- DataInputStream

FileOutputStream

该类用来创建一个文件并向文件中写数据。

如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。

有两个构造方法可以用来创建FileOutputStream 对象。

使用字符串类型的文件名来创建一个输出流对象：

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用File()方法来创建一个文件对象：

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

创建OutputStream 对象完成后，就可以使用下面的方法来写入流或者进行其他的流操作。

序号	方法及描述
public void close() throws IOException{}	关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。
protected void finalize()throws IOException {}	这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。
public void write(int w)throws IOException{}	这个方法把指定的字节写到输出流中。
public void write(byte[] w)	把指定数组中w.length长度的字节写到OutputStream中。

除了OutputStream外，还有一些其他的输出流，更多的细节参考下面链接：

- ByteArrayOutputStream
- DataOutputStream

实例

下面是一个演示InputStream和OutputStream用法的例子：

```
import java.io.*;

public class fileStreamTest{

   public static void main(String args[]){

   try{
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("test.txt");
      for(int x=0; x < bWrite.length ; x++){
         os.write( bWrite[x] ); // writes the bytes
      }
      os.close();

      InputStream is = new FileInputStream("test.txt");
      int size = is.available();

      for(int i=0; i< size; i++){
         System.out.print((char)is.read() + "  ");
      }
      is.close();
   }catch(IOException e){
      System.out.print("Exception");
   }
   }
}
```

上面的程序首先创建文件test.txt，并把给定的数字以二进制形式写进该文件，同时输出到控制台上。

以上代码由于是二进制写入，可能存在乱码，你可以使用以下代码实例来解决乱码问题：

```
//文件名 :fileStreamTest2.java
import java.io.*;

public class fileStreamTest2{
    public static void main(String[] args) throws IOException {

        File f = new File("a.txt");
        FileOutputStream fop = new FileOutputStream(f);
        // 构建FileOutputStream对象,文件不存在会自动新建

        OutputStreamWriter writer = new OutputStreamWriter(fop, "UTF-8");
        // 构建OutputStreamWriter对象,参数可以指定编码,默认为操作系统默认编码,windows上是gbk

        writer.append("中文输入");
        // 写入到缓冲区

        writer.append("\r\n");
        //换行

        writer.append("English");
        // 刷新缓存冲,写入到文件,如果下面已经没有写入的内容了,直接close也会写入

        writer.close();
        //关闭写入流,同时会把缓冲区内容写入文件,所以上面的注释掉

        fop.close();
        // 关闭输出流,释放系统资源

        FileInputStream fip = new FileInputStream(f);
        // 构建FileInputStream对象

        InputStreamReader reader = new InputStreamReader(fip, "UTF-8");
        // 构建InputStreamReader对象,编码与写入相同

        StringBuffer sb = new StringBuffer();
        while (reader.ready()) {
            sb.append((char) reader.read());
            // 转成char加到StringBuffer对象中
        }
        System.out.println(sb.toString());
        reader.close();
        // 关闭读取流

        fip.close();
        // 关闭输入流,释放系统资源

    }
}
```

文件和I/O

还有一些关于文件和I/O的类，我们也需要知道：

- File Class(类)
- FileReader Class(类)
- FileWriter Class(类)

Java中的目录

创建目录：

File类中有两个方法可以用来创建文件夹：

- **mkdir( )**方法创建一个文件夹，成功则返回true，失败则返回false。失败表明File对象指定的路径已经存在，或者由于整个路径还不存在，该文件夹不能被创建。
- **mkdirs()**方法创建一个文件夹和它的所有父文件夹。

下面的例子创建 "/tmp/user/java/bin"文件夹：

```
import java.io.File;

public class CreateDir {
   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);
      // 现在创建目录
      d.mkdirs();
  }
}
```

编译并执行上面代码来创建目录"/tmp/user/java/bin"。

注意：Java在UNIX和Windows自动按约定分辨文件路径分隔符。如果你在Windows版本的Java中使用分隔符(/)，路径依然能够被正确解析。

读取目录

一个目录其实就是一个File对象，它包含其他文件和文件夹。

如果创建一个File对象并且它是一个目录，那么调用isDirectory( )方法会返回true。

可以通过调用该对象上的list()方法，来提取它包含的文件和文件夹的列表。

下面展示的例子说明如何使用list()方法来检查一个文件夹中包含的内容：

```
import java.io.File;

public class DirList {
   public static void main(String args[]) {
      String dirname = "/tmp";
      File f1 = new File(dirname);
      if (f1.isDirectory()) {
         System.out.println( "Directory of " + dirname);
         String s[] = f1.list();
         for (int i=0; i < s.length; i++) {
            File f = new File(dirname + "/" + s[i]);
            if (f.isDirectory()) {
               System.out.println(s[i] + " is a directory");
            } else {
               System.out.println(s[i] + " is a file");
            }
         }
      } else {
         System.out.println(dirname + " is not a directory");
    }
  }
}
```

以上实例编译运行结果如下：

```
Directory of /tmp
bin is a directory
lib is a directory
demo is a directory
test.txt is a file
README is a file
index.html is a file
include is a directory
```

Java 异常处理

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。

比如说，你的代码少了一个分号，那么运行出来结果是提示是错误java.lang.Error；如果你用System.out.println(11/0)，那么你是因为你用0做了除数，会抛出java.lang.ArithmeticException的异常。

异常发生的原因有很多，通常包含以下几大类：

- 用户输入了非法数据。
- 要打开的文件不存在。
- 网络通信时连接中断，或者JVM内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。-

要理解Java异常处理是如何工作的，你需要掌握以下三种类型的异常：

- 检查性异常：最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
- 运行时异常： 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
- 错误： 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

Exception类的层次

所有的异常类是从java.lang.Exception类继承的子类。

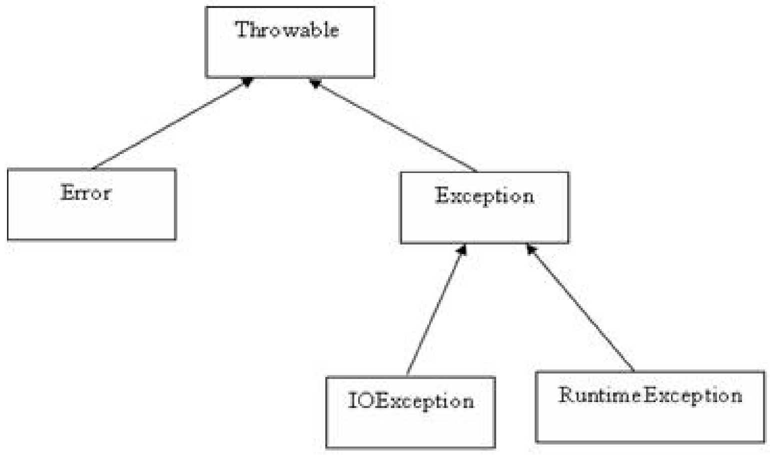
Exception类是Throwable类的子类。除了Exception类外，Throwable还有一个子类Error 。

Java程序通常不捕获错误。错误一般发生在严重故障时，它们在Java程序处理的范畴之外。

Error用来指示运行时环境发生的错误。

例如，JVM内存溢出。一般地，程序不会从错误中恢复。

异常类有两个主要的子类：IOException类和RuntimeException类。



在Java 内置类中(接下来会说明)，有大部分常用检查性和非检查性异常。

Java 内置异常类

Java 语言定义了一些异常类在java.lang标准包中。

标准运行时异常类的子类是最常见的异常类。由于java.lang包是默认加载到所有的Java程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。

Java根据各个类库也定义了一些其他的异常，下面的表中列出了Java的非检查性异常。

异常	描述
ArithmeticException	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
ArrayIndexOutOfBoundsException	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
ArrayStoreException	试图将错误类型的对象存储到一个对象数组时抛出的异常。
ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IllegalArgumentException	抛出的异常表明向方法传递了一个不合法或不正确的参数。
IllegalMonitorStateException	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。
IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 `null` 时，抛出该异常
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 `String` 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

下面的表中列出了Java定义在java.lang包中的检查性异常类。

异常	描述
ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedException	当调用 `Object` 类中的 `clone` 方法克隆对象，但该对象的类无法实现 `Cloneable` 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 `Class` 类中的 `newInstance` 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

异常方法

下面的列表是Throwable 类的主要方法:

方法	说明
public String getMessage()	返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。
public Throwable getCause()	返回一个Throwable 对象代表异常原因。
public String toString()	使用getMessage()的结果返回类的串级名字。
public void printStackTrace()	打印toString()结果和栈层次到System.err，即错误输出流。
public StackTraceElement [] getStackTrace()	返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。
public Throwable fillInStackTrace()	用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。

捕获异常

使用try和catch关键字可以捕获异常。try/catch代码块放在异常可能发生的地方。

try/catch代码块中的代码称为保护代码，使用 try/catch的语法如下：

```
try
{
   // 程序代码
}catch(ExceptionName e1)
{
   //Catch 块
}
```

Catch语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，try后面的catch块就会被检查。

如果发生的异常包含在catch块中，异常会被传递到该catch块，这和传递一个参数到方法是一样。

实例

下面的例子中声明有两个元素的一个数组，当代码试图访问数组的第三个元素的时候就会抛出一个异常。

```
// 文件名 : ExcepTest.java
import java.io.*;
public class ExcepTest{

   public static void main(String args[]){
      try{
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

以上代码编译运行输出结果如下：

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

多重捕获块

一个try代码块后面跟随多个catch代码块的情况就叫多重捕获。

多重捕获块的语法如下所示：

```
try{
   // 程序代码
}catch(异常类型1 异常的变量名1){
   // 程序代码
}catch(异常类型2 异常的变量名2){
   // 程序代码
}catch(异常类型2 异常的变量名2){
   // 程序代码
}
```

上面的代码段包含了3个catch块。

可以在ry语句后面添加任意数量的catch块。

如果保护代码中发生异常，异常被抛给第一个catch块。

如果抛出异常的数据类型与ExceptionType1匹配，它在这里就会被捕获。

如果不匹配，它会被传递给第二个catch块。

如此，直到异常被捕获或者通过所有的catch块。

实例

该实例展示了怎么使用多重try/catch。

```
try
{
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) //Not valid!
{
   f.printStackTrace();
   return -1;
}
```

throws/throw关键字：

如果一个方法没有捕获一个检查性异常，那么该方法必须使用throws 关键字来声明。throws 关键字放在方法签名的尾部。

也可以使用throw关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

下面方法的声明抛出一个RemoteException异常：

```
import java.io.*;
public class className
{
   public void deposit(double amount) throws RemoteException
   {
      // Method implementation
      throw new RemoteException();
   }
   //Remainder of class definition
}
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

例如，下面的方法声明抛出RemoteException和InsufficientFundsException：

```
import java.io.*;
public class className
{
   public void withdraw(double amount) throws RemoteException,
                              InsufficientFundsException
   {
       // Method implementation
   }
   //Remainder of class definition
}
```

finally关键字

finally关键字用来创建在try代码块后面执行的代码块。

无论是否发生异常，finally代码块中的代码总会被执行。

在finally代码块中，可以运行清理类型等收尾善后性质的语句。

finally代码块出现在catch代码块最后，语法如下：

```
try{
   // 程序代码
}catch(异常类型1 异常的变量名1){
   // 程序代码
}catch(异常类型2 异常的变量名2){
   // 程序代码
}finally{
   // 程序代码
}
```

实例

```
public class ExcepTest{

   public static void main(String args[]){
      int a[] = new int[2];
      try{
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      finally{
         a[0] = 6;
         System.out.println("First element value: " +a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

以上实例编译运行结果如下：

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

注意下面事项：

- catch不能独立于try存在。
- 在try/catch后面添加finally块并非强制性要求的。
- try代码后不能既没catch块也没finally块。
- try, catch, finally块之间不能添加任何代码。

声明自定义异常

在Java中你可以自定义异常。编写自己的异常类时需要记住下面的几点。

- 所有异常都必须是Throwable的子类。
- 如果希望写一个检查性异常类，则需要继承Exception类。
- 如果你想写一个运行时异常类，那么需要继承RuntimeException 类。

可以像下面这样定义自己的异常类：

```
class MyException extends Exception{
}
```

只继承Exception 类来创建的异常类是检查性异常类。

下面的InsufficientFundsException类是用户定义的异常类，它继承自Exception。

一个异常类和其它任何类一样，包含有变量和方法。

实例

```
// 文件名InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

为了展示如何使用我们自定义的异常类，

在下面的CheckingAccount 类中包含一个withdraw()方法抛出一个InsufficientFundsException异常。

```
// 文件名称 CheckingAccount.java
import java.io.*;

public class CheckingAccount
{
   private double balance;
   private int number;
   public CheckingAccount(int number)
   {
      this.number = number;
   }
   public void deposit(double amount)
   {
      balance += amount;
   }
   public void withdraw(double amount) throws
                              InsufficientFundsException
   {
      if(amount <= balance)
      {
         balance -= amount;
      }
      else
      {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }
   public double getBalance()
   {
      return balance;
   }
   public int getNumber()
   {
      return number;
   }
}
```

下面的BankDemo程序示范了如何调用CheckingAccount类的deposit() 和withdraw()方法。

```
//文件名称 BankDemo.java
public class BankDemo
{
   public static void main(String [] args)
   {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);
      try
      {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      }catch(InsufficientFundsException e)
      {
         System.out.println("Sorry, but you are short $"
                                  + e.getAmount());
         e.printStackTrace();
      }
    }
}
```

编译上面三个文件，并运行程序BankDemo，得到结果如下所示：

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

通用异常

在Java中定义了两种类型的异常和错误。

- **JVM(Java**虚拟机)异常：**由JVM抛出的异常或错误。例如：NullPointerException类，ArrayIndexOutOfBoundsException类，ClassCastException类。
- 程序级异常：由程序或者API程序抛出的异常。例如IllegalArgumentException类，IllegalStateException类。

Java 面向对象

Java 继承

继承是java面向对象编程技术的一块基石，因为它允许创建分等级层次的类。继承可以理解为一个对象从另一个对象获取属性的过程。

如果类A是类B的父类，而类B是类C的父类，我们也称C是A的子类，类C是从类A继承而来的。在Java中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类

继承中最常使用的两个关键字是extends和implements。

这两个关键字的使用决定了一个对象和另一个对象是否是IS-A(是一个)关系。

通过使用这两个关键字，我们能实现一个对象获取另一个对象的属性。

所有Java的类均是由java.lang.Object类继承而来的，所以Object是所有类的祖先类，而除了Object外，所有类必须有一个父类。

通过过extends关键字可以申明一个类是继承另外一个类而来的，一般形式如下：

```
// A.java
public class A {
    private int i;
    protected int j;

    public void func() {

    }
}

// B.java
public class B extends A {
}
```

以上的代码片段说明，B由A继承而来的，B是A的子类。而A是Object的子类，这里可以不显示地声明。

作为子类，B的实例拥有A所有的成员变量，但对于private的成员变量B却没有访问权限，这保障了A的封装性。

IS-A关系

IS-A就是说:一个对象是另一个对象的一个分类。

下面是使用关键字extends实现继承。

```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

基于上面的例子，以下说法是正确的：

- Animal类是Mammal类的父类。
- Animal类是Reptile类的父类。
- Mammal类和Reptile类是Animal类的子类。
- Dog类既是Mammal类的子类又是Animal类的子类。

分析以上示例中的IS-A关系，如下：

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal

因此 : Dog IS-A Animal

通过使用关键字**extends**，子类可以继承父类的除private属性外所有的属性。

我们通过使用instanceof 操作符，能够确定Mammal IS-A Animal

实例

```
public class Dog extends Mammal{

   public static void main(String args[]){

      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

以上实例编译运行结果如下：

```
true
true
true
```

介绍完**extends**关键字之后，我们再来看下**implements**关键字是怎样使用来表示IS-A关系。

Implements关键字使用在类继承接口的情况下， 这种情况不能使用关键字**extends**。

实例

```
public interface Animal {}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

instanceof 关键字

可以使用 **instanceof** 运算符来检验Mammal和dog对象是否是Animal类的一个实例。

```
interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{
   public static void main(String args[]){

      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

以上实例编译运行结果如下：

```
true
true
true
```

HAS-A 关系

HAS-A代表类和它的成员之间的从属关系。这有助于代码的重用和减少代码的错误。

例子

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
    private Speed sp;
}
```

Van类和Speed类是HAS-A关系(Van有一个Speed)，这样就不用将Speed类的全部代码粘贴到Van类中了，并且Speed类也可以重复利用于多个应用程序。

在面向对象特性中，用户不必担心类的内部怎样实现。

Van类将实现的细节对用户隐藏起来，因此，用户只需要知道怎样调用Van类来完成某一功能，而不必知道Van类是自己来做还是调用其他类来做这些工作。

Java只支持单继承，也就是说，一个类不能继承多个类。

下面的做法是不合法的：

```
public class extends Animal, Mammal{}
```

Java只支持单继承（继承基本类和抽象类），但是我们可以用接口来实现（多继承接口来实现）,脚本结构如：

```
public class Apple extends Fruit implements Fruit1, Fruit2{}
```

一般我们继承基本类和抽象类用extends关键字，实现接口类的继承用implements关键字。

Java 重写(Override)与重载(Overload)

重写(Override)

重写是子类对父类的允许访问的方法的实现过程进行重新编写！返回值和形参都不能改变。即外壳不变，核心重写！

重写的好处在于子类可以根据需要，定义特定于自己的行为。

也就是说子类能够根据需要实现父类的方法。

在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

```
class Animal{

   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{

   public void move(){
      System.out.println("狗可以跑和走");
   }
}

public class TestDog{

   public static void main(String args[]){
      Animal a = new Animal(); // Animal 对象
      Animal b = new Dog(); // Dog 对象

      a.move();// 执行 Animal 类的方法

      b.move();//执行 Dog 类的方法
   }
}
```

以上实例编译运行结果如下：

```
动物可以移动
狗可以跑和走
```

在上面的例子中可以看到，尽管b属于Animal类型，但是它运行的是Dog类的move方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为Animal类中存在move方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

```
class Animal{

   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{

   public void move(){
      System.out.println("狗可以跑和走");
   }
   public void bark(){
      System.out.println("狗可以吠叫");
   }
}

public class TestDog{

   public static void main(String args[]){
      Animal a = new Animal(); // Animal 对象
      Animal b = new Dog(); // Dog 对象

      a.move();// 执行 Animal 类的方法
      b.move();//执行 Dog 类的方法
      b.bark();
   }
}
```

以上实例编译运行结果如下：

```
TestDog.java:30: cannot find symbol
symbol  : method bark()
location: class Animal
                b.bark();
                 ^
```

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

方写重写的规则

- 参数列表必须完全与被重写方法的相同；
- 返回类型必须完全与被重写方法的返回类型相同；
- 访问权限不能比父类中被重写的方法的访问权限更高。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。
- 父类的成员方法只能被它的子类重写。
- 声明为final的方法不能被重写。
- 声明为static的方法不能被重写，但是能够被再次声明。
- 如果一个方法不能被继承，那么该方法不能被重写。
- 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。

- 子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。
- 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
- 构造方法不能被重写。
- 如果不能继承一个方法，则不能重写这个方法。

Super关键字的使用

当需要在子类中调用父类的被重写方法时，要使用super关键字。

```
class Animal{

   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{

   public void move(){
      super.move(); // 应用super类的方法
      System.out.println("狗可以跑和走");
   }
}

public class TestDog{

   public static void main(String args[]){

      Animal b = new Dog(); /
      b.move(); //执行 Dog类的方法

   }
}
```

以上实例编译运行结果如下：

```
动物可以移动
狗可以跑和走
```

重载(Overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型呢？可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

只能重载构造函数

重载规则

- 被重载的方法必须改变参数列表；
- 被重载的方法可以改变返回类型；
- 被重载的方法可以改变访问修饰符；
- 被重载的方法可以声明新的或更广的检查异常；
- 方法能够在同一个类中或者在一个子类中被重载。

实例

```
public class Overloading {

    public int test(){
        System.out.println("test1");
        return 1;
    }

    public void test(int a){
        System.out.println("test2");
    }

    //以下两个参数类型顺序不同
    public String test(int a,String s){
        System.out.println("test3");
        return "returntest3";
    }

    public String test(String s,int a){
        System.out.println("test4");
        return "returntest4";
    }

    public static void main(String[] args){
        Overloading o = new Overloading();
        System.out.println(o.test());
        o.test(1);
        System.out.println(o.test(1,"test3"));
        System.out.println(o.test("test4",1));
    }
```

重写与重载之间的区别

区别点	重载方法	重写方法
参数列表	必须修改	一定不能修改
返回类型	可以修改	一定不能修改
异常	可以修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问	可以修改	一定不能做更严格的限制（可以降低限制）

Java 多态

多态是同一个行为具有多个不同表现形式或形态的能力。

多态性是对象多种表现形式的体现。

比如我们说"宠物"这个对象，它就有很多不同的表达或实现，比如有小猫、小狗、蜥蜴等等。那么我到宠物店说"请给我一只宠物"，服务员给我小猫、小狗或者蜥蜴都可以，我们就说"宠物"这个对象就具备多态性。

接下来让我们通过实例来了解Java的多态。

例子

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

因为Deer类具有多重继承，所以它具有多态性。以上实例解析如下：

- 一个 Deer IS-A（是一个） Animal
- 一个 Deer IS-A（是一个） Vegetarian
- 一个 Deer IS-A（是一个） Deer
- 一个 Deer IS-A（是一个）Object

在Java中，所有的对象都具有多态性，因为任何对象都能通过IS-A测试的类型和Object类。

访问一个对象的唯一方法就是通过引用型变量。

引用型变量只能有一种类型，一旦被声明，引用型变量的类型就不能被改变了。

引用型变量不仅能够被重置为其他对象，前提是这些对象没有被声明为final。还可以引用和它类型相同的或者相兼容的对象。它可以声明为类类型或者接口类型。

当我们将引用型变量应用于Deer对象的引用时，下面的声明是合法的：

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

所有的引用型变量d,a,v,o都指向堆中相同的Deer对象。

虚方法

我们将介绍在Java中，当设计类时，被重载的方法的行为怎样影响多态性。

我们已经讨论了方法的重载，也就是子类能够重载父类的方法。

当子类对象调用重载的方法时，调用的是子类的方法，而不是父类中被重载的方法。

要想调用父类中被重载的方法，则必须使用关键字super。

```
/* 文件名 : Employee.java */
public class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
      return name;
   }
   public String getAddress()
   {
      return address;
   }
   public void setAddress(String newAddress)
   {
      address = newAddress;
   }
   public int getNumber()
   {
     return number;
   }
}
```

假设下面的类继承Employee类：

```
/* 文件名 : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double
      salary)
   {
       super(name, address, number);
       setSalary(salary);
   }
   public void mailCheck()
   {
       System.out.println("Within mailCheck of Salary class ");
       System.out.println("Mailing check to " + getName()
       + " with salary " + salary);
   }
   public double getSalary()
   {
       return salary;
   }
   public void setSalary(double newSalary)
   {
       if(newSalary >= 0.0)
       {
          salary = newSalary;
       }
   }
   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

现在我们仔细阅读下面的代码，尝试给出它的输出结果：

```
/* 文件名 : VirtualDemo.java */
public class VirtualDemo
{
   public static void main(String [] args)
   {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

以上实例编译运行结果如下：

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

例子中，我们实例化了两个Salary对象。一个使用Salary引用s，另一个使用Employee引用。

编译时，编译器检查到mailCheck()方法在Salary类中的声明。

在调用s.mailCheck()时，Java虚拟机(JVM)调用Salary类的mailCheck()方法。

因为e是Employee的引用，所以调用e的mailCheck()方法则有完全不同的结果。

当编译器检查e.mailCheck()方法时，编译器检查到Employee类中的mailCheck()方法。

在编译的时候，编译器使用Employee类中的mailCheck()方法验证该语句， 但是在运行的时候，Java虚拟机(JVM)调用的是Salary类中的mailCheck()方法。

该行为被称为虚拟方法调用，该方法被称为虚拟方法。

Java中所有的方法都能以这种方式表现，借此，重写的方法能在运行时调用，不管编译的时候源代码中引用变量是什么数据类型。

Java 抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。也是因为这个原因，通常在设计阶段决定要不要设计抽象类。

父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。

抽象类

在Java语言中使用abstract class来定义抽象类。如下实例：

```
/* 文件名 : Employee.java */
public abstract class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }
   public double computePay()
   {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
      return name;
   }
   public String getAddress()
   {
      return address;
   }
   public void setAddress(String newAddress)
   {
      address = newAddress;
   }
   public int getNumber()
   {
     return number;
   }
}
```

注意到该Employee类没有什么不同，尽管该类是抽象类，但是它仍然有3个成员变量，7个成员方法和1个构造方法。 现在如果你尝试如下的例子：

```
/* 文件名 : AbstractDemo.java */
public class AbstractDemo
{
   public static void main(String [] args)
   {
      /* 以下是不允许的，会引发错误 */
      Employee e = new Employee("George W.", "Houston, TX", 43);

      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

当你尝试编译AbstractDemo类时，会产生如下错误：

```
Employee.java:46: Employee is abstract; cannot be instantiated
      Employee e = new Employee("George W.", "Houston, TX", 43);
                   ^
1 error
```

继承抽象类

我们能通过一般的方法继承Employee类：

```
/* 文件名 : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double
      salary)
   {
       super(name, address, number);
       setSalary(salary);
   }
   public void mailCheck()
   {
       System.out.println("Within mailCheck of Salary class ");
       System.out.println("Mailing check to " + getName()
       + " with salary " + salary);
   }
   public double getSalary()
   {
       return salary;
   }
   public void setSalary(double newSalary)
   {
       if(newSalary >= 0.0)
       {
          salary = newSalary;
       }
   }
   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

尽管我们不能实例化一个Employee类的对象，但是如果我们实例化一个Salary类对象，该对象将从Employee类继承3个成员变量和7个成员方法。

```
/* 文件名 : AbstractDemo.java */
public class AbstractDemo
{
   public static void main(String [] args)
   {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();

      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

以上程序编译运行结果如下：

```
Constructing an Employee
Constructing an Employee
Call mailCheck using  Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.
```

抽象方法

如果你想设计这样一个类，该类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么你可以在父类中声明该方法为抽象方法。

Abstract关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

抽象方法没有定义，方法名后面直接跟一个分号，而不是花括号。

```
public abstract class Employee
{
   private String name;
   private String address;
   private int number;

   public abstract double computePay();

   //其余代码
}
```

声明抽象方法会造成以下两个结果：

- 如果一个类包含抽象方法，那么该类必须是抽象类。
- 任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

继承抽象方法的子类必须重载该方法。否则，该子类也必须声明为抽象类。最终，必须有子类实现该抽象方法，否则，从最初的父类到最终的子类都不能用来实例化对象。

如果Salary类继承了Employee类，那么它必须实现computePay()方法：

```
/* 文件名 : Salary.java */
public class Salary extends Employee
{
   private double salary; // Annual salary

   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }

   //其余代码
}
```

java 封装

在面向对象程式设计方法中，封装（英语：Encapsulation）是指，一种将抽象性函式接口的实作细节部份包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

实例

让我们来看一个java封装类的例子：

```
/* 文件名: EncapTest.java */
public class EncapTest{

   private String name;
   private String idNum;
   private int age;

   public int getAge(){
      return age;
   }

   public String getName(){
      return name;
   }

   public String getIdNum(){
      return idNum;
   }

   public void setAge( int newAge){
      age = newAge;
   }

   public void setName(String newName){
      name = newName;
   }

   public void setIdNum( String newId){
      idNum = newId;
   }
}
```

以上实例中public方法是外部类访问该类成员变量的入口。

通常情况下，这些方法被称为getter和setter方法。

因此，任何要访问类中私有成员变量的类都要通过这些getter和setter方法。

通过如下的例子说明EncapTest类的变量怎样被访问：

```
/* F文件名 : RunEncap.java */
public class RunEncap{

   public static void main(String args[]){
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName()+
                             " Age : "+ encap.getAge());
    }
}
```

以上代码编译运行结果如下:

```
Name : James Age : 20
```

Java 接口

接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在Java中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在.java结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在.class结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别：

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了static和final变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多重继承。

接口的声明

接口的声明语法格式如下：

```
[可见度] interface 接口名称 [extends 其他的类名] {
        // 声明变量
        // 抽象方法
}
```

Interface关键字用来声明一个接口。下面是接口声明的一个简单例子。

```
/* 文件名 : NameOfInterface.java */
import java.lang.*;
//引入包

public interface NameOfInterface
{
   //任何类型 final, static 字段
   //抽象方法
}
```

接口有以下特性：

- 接口是隐式抽象的，当声明一个接口的时候，不必使用**abstract**关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要**abstract**关键子。
- 接口中的方法都是公有的。

实例

```
/* 文件名 : Animal.java */
interface Animal {

   public void eat();
   public void travel();
}
```

接口的实现

当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。

类使用implements关键字实现接口。在类声明中，Implements关键字放在class声明后面。

实现一个接口的语法，可以使用这个公式：

```
... implements 接口名称[, 其他接口, 其他接口..., ...] ...
```

实例

```
/* 文件名 : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

以上实例编译运行结果如下:

```
Mammal eats
Mammal travels
```

重写接口中声明的方法时，需要注意以下规则：

- 类在实现接口的方法时，不能抛出强制性异常，只能在接口中，或者继承接口的抽象类中抛出该强制性异常。
- 类在重写方法时要保持一致的方法名，并且应该保持相同或者相兼容的返回值类型。
- 如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口的时候，也要注意一些规则：

- 一个类可以同时实现多个接口。
- 一个类只能继承一个类，但是能实现多个接口。
- 一个接口能继承另一个接口，这和类之间的继承比较相似。

接口的继承

一个接口能继承另一个接口，和类之间的继承方式比较相似。接口的继承使用extends关键字，子接口继承父接口的方法。

下面的Sports接口被Hockey和Football接口继承：

```
// 文件名: Sports.java
public interface Sports
{
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// 文件名: Football.java
public interface Football extends Sports
{
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// 文件名: Hockey.java
public interface Hockey extends Sports
{
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

Hockey接口自己声明了四个方法，从Sports接口继承了两个方法，这样，实现Hockey接口的类需要实现六个方法。

相似的，实现Football接口的类需要实现五个方法，其中两个来自于Sports接口。

接口的多重继承

在Java中，类的多重继承是不合法，但接口允许多重继承，。

在接口的多重继承中extends关键字只需要使用一次，在其后跟着继承接口。 如下所示：

```
public interface Hockey extends Sports, Event
```

以上的程序片段是合法定义的子接口，与类不同的是，接口允许多重继承，而 Sports及 Event 可能定义或是继承相同的方法

标记接口

最常用的继承接口是没有包含任何方法的接口。

标识接口是没有任何方法和属性的接口.它仅仅表明它的类属于一个特定的类型,供其他代码来测试允许做一些事情。

标识接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：java.awt.event包中的MouseListener接口继承的java.util.EventListener接口定义如下：

```
package java.util;
public interface EventListener
{}
```

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

- 建立一个公共的父接口：

  正如EventListener接口，这是由几十个其他接口扩展的Java API，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了EventListener接口，Java虚拟机(JVM)就知道该接口将要被用于一个事件的代理方案。

- 向一个类添加数据类型：

  这种情况是标记接口最初的目的，实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

Java 包(package)

为了更好地组织类，Java提供了包机制，用于区别类名的命名空间。

包的作用

- 1 把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。
- 2 如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。
- 3 包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。

Java使用包（package）这种机制是为了防止命名冲突，访问控制，提供搜索和定位类（class）、接口、枚举（enumerations）和注释（annotation）等。

包语句的语法格式为：

```
package pkg1[. pkg2[. pkg3…]];
```

例如,一个Something.java 文件它的内容

```
package net.java.util
public class Something{
   ...
}
```

那么它的路径应该是 net/java/Something.java 这样保存的。 package(包)的作用是把不同的java程序分类保存，更方便的被其他java程序调用。

一个包（package）可以定义为一组相互联系的类型（类、接口、枚举和注释），为这些类型提供访问保护和命名空间管理的功能。

以下是一些Java中的包：

- java.lang-打包基础的类
- java.io-包含输入输出功能的函数

开发者可以自己把一组类和接口等打包，并定义自己的package。而且在实际开发中这样做是值得提倡的，当你自己完成类的实现之后，将相关的类分组，可以让其他的编程者更容易地确定哪些类、接口、枚举和注释等是相关的。

由于package创建了新的命名空间（namespace），所以不会跟其他package中的任何名字产生命名冲突。使用包这种机制，更容易实现访问控制，并且让定位相关类更加简单。

创建包

创建package的时候，你需要为这个package取一个合适的名字。之后，如果其他的一个源文件包含了这个包提供的类、接口、枚举或者注释类型的时候，都必须将这个package的声明放在这个源文件的开头。

包声明应该在源文件的第一行，每个源文件只能有一个包声明，这个文件中的每个类型都应用于它。

如果一个源文件中没有使用包声明，那么其中的类，函数，枚举，注释等将被放在一个无名的包（unnamed package）中。

例子

让我们来看一个例子，这个例子创建了一个叫做animals的包。通常使用小写的字母来命名避免与类、接口名字的冲突。

在animals包中加入一个接口（interface）：

```
/* 文件名: Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

接下来，在同一个包中加入该接口的实现：

```
package animals;

/* 文件名 : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

然后，编译这两个文件，并把他们放在一个叫做animals的子目录中。 用下面的命令来运行：

```
$ mkdir animals
$ cp Animal.class  MammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travel
```

import关键字

为了能够使用某一个包的成员，我们需要在 Java 程序中明确导入该包。使用"import"语句可完成此功能。

在 java 源文件中 import 语句应位于 package 语句之后，所有类的定义之前，可以没有，也可以有多条，其语法格式为：

```
import package1[.package2…].(classname|*);
```

如果在一个包中，一个类想要使用本包中的另一个类，那么该包名可以省略。

例子

下面的payroll包已经包含了Employee类，接下来向payroll包中添加一个Boss类。Boss类引用Employee类的时候可以不用使用payroll前缀，Boss类的实例如下。

```
package payroll;

public class Boss
{
   public void payEmployee(Employee e)
   {
      e.mailCheck();
   }
}
```

如果Boss类不在payroll包中又会怎样？Boss类必须使用下面几种方法之一来引用其他包中的类

使用类全名描述，例如：

```
payroll.Employee
```

用import关键字引入，使用通配符"*"

```
import payroll.*;
```

使用import关键字引入Employee类

```
import payroll.Employee;
```

注意：

类文件中可以包含任意数量的import声明。import声明必须在包声明之后，类声明之前。

package的目录结构

类放在包中会有两种主要的结果：

- 包名成为类名的一部分，正如我们前面讨论的一样。
- 包名必须与相应的字节码所在的目录结构相吻合。

下面是管理你自己java中文件的一种简单方式：

将类、接口等类型的源码放在一个文本中，这个文件的名字就是这个类型的名字，并以.java作为扩展名。例如：

```
// 文件名 :  Car.java

package vehicle;

public class Car {
   // 类实现
}
```

接下来，把源文件放在一个目录中，这个目录要对应类所在包的名字。

```
....\vehicle\Car.java
```

现在，正确的类名和路径将会是如下样子：

- 类名 -> vehicle.Car
- 路径名 -> vehicle\Car.java (in windows)

通常，一个公司使用它互联网域名的颠倒形式来作为它的包名.例如：互联网域名是apple.com，所有的包名都以com.apple开头。包名中的每一个部分对应一个子目录。

例如：这个公司有一个com.apple.computers的包，这个包包含一个叫做Dell.java的源文件，那么相应的，应该有如下面的一连串子目录：

```
....\com\apple\computers\Dell.java
```

编译的时候，编译器为包中定义的每个类、接口等类型各创建一个不同的输出文件，输出文件的名字就是这个类型的名字，并加上.class作为扩展后缀。 例如：

```
// 文件名: Dell.java

package com.apple.computers;
public class Dell{

}
class Ups{

}
```

现在，我们用-d选项来编译这个文件，如下：

```
$javac -d . Dell.java
```

这样会像下面这样放置编译了的文件：

```
.\com\apple\computers\Dell.class.\com\apple\computers\Ups.class
```

你可以像下面这样来导入所有 \com\apple\computers\中定义的类、接口等：

```
import com.apple.computers.*;
```

编译之后的.class文件应该和.java源文件一样，它们放置的目录应该跟包的名字对应起来。但是，并不要求.class文件的路径跟相应的.java的路径一样。你可以分开来安排源码和类的目录。

```
<path-one>\sources\com\apple\computers\Dell.java
<path-two>\classes\com\apple\computers\Dell.class
```

这样，你可以将你的类目录分享给其他的编程人员，而不用透露自己的源码。用这种方法管理源码和类文件可以让编译器和java虚拟机（JVM）可以找到你程序中使用的所有类型。

类目录的绝对路径叫做class path。设置在系统变量CLASSPATH中。编译器和java虚拟机通过将package名字加到class path后来构造.class文件的路径。

<path- two>\classes是class path，package名字是com.apple.computers,而编译器和JVM会在 <path-two>\classes\com\apple\compters中找.class文件。

一个class path可能会包含好几个路径。多路径应该用分隔符分开。默认情况下，编译器和JVM查找当前目录。JAR文件按包含Java平台相关的类，所以他们的目录默认放在了class path中。

设置CLASSPATH系统变量

用下面的命令显示当前的CLASSPATH变量：

- Windows平台（DOS 命令行下）-> C:\> set CLASSPATH
- UNIX平台（Bourne shell下）-> % echo $CLASSPATH

删除当前CLASSPATH变量内容：

- Windows平台（DOS 命令行下）-> C:\> set CLASSPATH=
- UNIX平台（Bourne shell下）-> % unset CLASSPATH; export CLASSPATH

设置CLASSPATH变量:

- Windows平台（DOS 命令行下）-> set CLASSPATH=C:\users\jack\java\classes
- UNIX平台（Bourne shell下）-> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

Java 高级教程

Java 数据结构

Java工具包提供了强大的数据结构。在Java中的数据结构主要包括以下几种接口和类：

- 枚举（Enumeration）
- 位集合（BitSet）
- 向量（Vector）
- 栈（Stack）
- 字典（Dictionary）
- 哈希表（Hashtable）
- 属性（Properties）

以上这些类是传统遗留的，在Java2中引入了一种新的框架-集合框架(Collection)，我们后面再讨论。

枚举（Enumeration）

枚举（Enumeration）接口虽然它本身不属于数据结构,但它在其他数据结构的范畴里应用很广。 枚举（The Enumeration）接口定义了一种从数据结构中取回连续元素的方式。

例如，枚举定义了一个叫nextElement 的方法，该方法用来得到一个包含多元素的数据结构的下一个元素。

关于枚举接口的更多信息，请参见枚举（Enumeration）。

位集合（BitSet）

位集合类实现了一组可以单独设置和清除的位或标志。

该类在处理一组布尔值的时候非常有用，你只需要给每个值赋值一"位"，然后对位进行适当的设置或清除，就可以对布尔值进行操作了。

关于该类的更多信息，请参见位集合（BitSet）。

向量（Vector）

向量（Vector）类和传统数组非常相似，但是Vector的大小能根据需要动态的变化。

和数组一样，Vector对象的元素也能通过索引访问。

使用Vector类最主要的好处就是在创建对象的时候不必给对象指定大小，它的大小会根据需要动态的变化。

关于该类的更多信息，请参见向量(Vector)

栈（Stack）

栈（Stack）实现了一个后进先出（LIFO）的数据结构。

你可以把栈理解为对象的垂直分布的栈，当你添加一个新元素时，就将新元素放在其他元素的顶部。

当你从栈中取元素的时候，就从栈顶取一个元素。换句话说，最后进栈的元素最先被取出。

关于该类的更多信息，请参见栈（Stack）。

字典（Dictionary）

字典（Dictionary） 类是一个抽象类，它定义了键映射到值的数据结构。

当你想要通过特定的键而不是整数索引来访问数据的时候，这时候应该使用Dictionary。

由于Dictionary类是抽象类，所以它只提供了键映射到值的数据结构，而没有提供特定的实现。

关于该类的更多信息，请参见字典（ Dictionary）。

哈希表（Hashtable）

Hashtable类提供了一种在用户定义键结构的基础上来组织数据的手段。

例如，在地址列表的哈希表中，你可以根据邮政编码作为键来存储和排序数据，而是通过人的名字。

哈希表键的具体含义完全取决于哈希表的使用情景和它包含的数据。

关于该类的更多信息，请参见哈希表（HashTable）。

属性（Properties）

Properties 继承于 Hashtable.Properties 类表示了一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。

Properties 类被许多Java类使用。例如，在获取环境变量时它就作为System.getProperties()方法的返回值。

关于该类的更多信息，请参见属性（Properties）。

Java Enumeration接口

Enumeration接口中定义了一些方法，通过这些方法可以枚举（一次获得一个）对象集合中的元素。

这种传统接口已被迭代器取代，虽然Enumeration 还未被遗弃，但在现代代码中已经被很少使用了。尽管如此，它还是使用在诸如Vector和Properties这些传统类所定义的方法中，除此之外，还用在一些API类，并且在应用程序中也广泛被使用。 下表总结了一些Enumeration声明的方法：

方法	描述
boolean hasMoreElements( )	测试此枚举是否包含更多的元素。
Object nextElement( )	如果此枚举对象至少还有一个可提供的元素，则返回此枚举的下一个元素。

实例

以下实例演示了Enumeration的使用：

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester {

   public static void main(String args[]) {
      Enumeration days;
      Vector dayNames = new Vector();
      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();
      while (days.hasMoreElements()){
         System.out.println(days.nextElement());
      }
   }
}
```

以上实例编译运行结果如下：

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Java Bitset类

一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。这和位向量（vector of bits）比较类似。

这是一个传统的类，但它在Java 2中被完全重新设计。

BitSet定义了两个构造方法。

第一个构造方法创建一个默认的对象：

```
BitSet()
```

第二个方法允许用户指定初始大小。所有位初始化为0。

```
BitSet(int size)
```

BitSet中实现了Cloneable接口中定义的方法如下表所列：

方法	描述
void and(BitSet bitSet)	对此目标位 set 和参数位 set 执行逻辑与操作。
void andNot(BitSet bitSet)	清除此 BitSet 中所有的位，其相应的位在指定的 BitSet 中已设置。
int cardinality( )	返回此 BitSet 中设置为 true 的位数。
void clear( )	将此 BitSet 中的所有位设置为 false。
void clear(int index)	将索引指定处的位设置为 false。
void clear(int startIndex, int endIndex)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为 false。
Object clone( )	复制此 BitSet，生成一个与之相等的新 BitSet。
boolean equals(Object bitSet)	将此对象与指定的对象进行比较。
void flip(int index)	将指定索引处的位设置为其当前值的补码。
void flip(int startIndex, int endIndex)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的每个位设置为其当前值的补码。
boolean get(int index)	返回指定索引处的位值。
BitSet get(int startIndex, int endIndex)	返回一个新的 BitSet，它由此 BitSet 中从 fromIndex（包括）到 toIndex（不包括）范围内的位组成。

boolean intersects(BitSet bitSet)	如果指定的 BitSet 中有设置为 true 的位，并且在此 BitSet 中也将其设置为 true，则返回 ture。
boolean isEmpty( )	如果此 BitSet 中没有包含任何设置为 true 的位，则返回 ture。
int length( )	返回此 BitSet 的"逻辑大小"：BitSet 中最高设置位的索引加 1。
int nextClearBit(int startIndex)	返回第一个设置为 false 的位的索引，这发生在指定的起始索引或之后的索引上。
int nextSetBit(int startIndex)	返回第一个设置为 true 的位的索引，这发生在指定的起始索引或之后的索引上。
void or(BitSet bitSet)	对此位 set 和位 set 参数执行逻辑或操作。
void set(int index)	将指定索引处的位设置为 true。
void set(int index, boolean v)	将指定索引处的位设置为指定的值。
void set(int startIndex, int endIndex)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为 true。
void set(int startIndex, int endIndex, boolean v)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为指定的值。
int size( )	返回此 BitSet 表示位值时实际使用空间的位数。
String toString( )	返回此位 set 的字符串表示形式。
void xor(BitSet bitSet)	对此位 set 和位 set 参数执行逻辑异或操作。

实例

下面的程序说明这个数据结构支持的几个方法：

```
import java.util.BitSet;

public class BitSetDemo {

  public static void main(String args[]) {
     BitSet bits1 = new BitSet(16);
     BitSet bits2 = new BitSet(16);

     // set some bits
     for(int i=0; i<16; i++) {
        if((i%2) == 0) bits1.set(i);
        if((i%5) != 0) bits2.set(i);
     }
     System.out.println("Initial pattern in bits1: ");
     System.out.println(bits1);
     System.out.println("\nInitial pattern in bits2: ");
     System.out.println(bits2);

     // AND bits
     bits2.and(bits1);
     System.out.println("\nbits2 AND bits1: ");
     System.out.println(bits2);

     // OR bits
     bits2.or(bits1);
     System.out.println("\nbits2 OR bits1: ");
     System.out.println(bits2);

     // XOR bits
     bits2.xor(bits1);
     System.out.println("\nbits2 XOR bits1: ");
     System.out.println(bits2);
  }
}
```

以上实例编译运行结果如下：

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

Java Vector 类

Vector类实现了一个动态数组。和ArrayList和相似，但是两者是不同的：

- Vector是同步访问的。
- Vector包含了许多传统的方法，这些方法不属于集合框架。

Vector主要用在事先不知道数组的大小，或者只是需要一个可以改变大小的数组的情况。

Vector类支持4种构造方法。

第一种构造方法创建一个默认的向量，默认大小为10：

```
Vector()
```

第二种构造方法创建指定大小的向量。

```
Vector(int size)
```

第三种构造方法创建指定大小的向量，并且增量用incr指定. 增量表示向量每次增加的元素数目。

```
Vector(int size,int incr)
```

第四中构造方法创建一个包含集合c元素的向量：

```
Vector(Collection c)
```

除了从父类继承的方法外Vector还定义了以下方法：

方法	描述
void add(int index, Object element)	在此向量的指定位置插入指定的元素。
boolean add(Object o)	将指定元素添加到此向量的末尾。
boolean addAll(Collection c)	将指定 Collection 中的所有元素添加到此向量的末尾，按照指定 collection 的迭代器所返回的顺序添加这些元素。
boolean addAll(int index, Collection c)	在指定位置将指定 Collection 中的所有元素插入到此向量中。
void addElement(Object obj)	将指定的组件添加到此向量的末尾，将其大小增加 1。

int capacity()	返回此向量的当前容量。	
void clear()	从此向量中移除所有元素。	
Object clone()	返回向量的一个副本。	
boolean contains(Object elem)	如果此向量包含指定的元素，则返回 true。	
boolean containsAll(Collection c)	如果此向量包含指定 Collection 中的所有元素，则返回 true。	
void copyInto(Object[] anArray)	将此向量的组件复制到指定的数组中。	
Object elementAt(int index)	返回指定索引处的组件。	
Enumeration elements()	返回此向量的组件的枚举。	
void ensureCapacity(int minCapacity)	增加此向量的容量（如有必要），以确保其至少能够保存最小容量参数指定的组件数。	
boolean equals(Object o)	比较指定对象与此向量的相等性。	
Object firstElement()	返回此向量的第一个组件（位于索引 0)	处的项）。
Object get(int index)	返回向量中指定位置的元素。	
int hashCode()	返回此向量的哈希码值。	
int indexOf(Object elem)	返回此向量中第一次出现的指定元素的索引，如果此向量不包含该元素，则返回 -1。	
int indexOf(Object elem, int index)	返回此向量中第一次出现的指定元素的索引，从 index 处正向搜索，如果未找到该元素，则返回 -1。	
void insertElementAt(Object obj, int index)	将指定对象作为此向量中的组件插入到指定的 index 处。	
boolean isEmpty()	测试此向量是否不包含组件。	
Object lastElement()	返回此向量的最后一个组件。	
int lastIndexOf(Object elem)	返回此向量中最后一次出现的指定元素的索引；如果此向量不包含该元素，则返回 -1。	
int lastIndexOf(Object elem, int index)	返回此向量中最后一次出现的指定元素的索引，从 index 处逆向搜索，如果未找到该元素，则返回 -1。	
Object remove(int index)	移除此向量中指定位置的元素。	
boolean remove(Object o)	移除此向量中指定元素的第一个匹配项，如果向量不包含该元素，则元素保持不变。	

boolean removeAll(Collection c)	从此向量中移除包含在指定 Collection 中的所有元素。
void removeAllElements()	从此向量中移除全部组件，并将其大小设置为零。
boolean removeElement(Object obj)	从此向量中移除变量的第一个（索引最小的）匹配项。
void removeElementAt(int index)	删除指定索引处的组件。
protected void removeRange(int fromIndex, int toIndex)	从此 List 中移除其索引位于 fromIndex（包括）与 toIndex（不包括）之间的所有元素。
boolean retainAll(Collection c)	在此向量中仅保留包含在指定 Collection 中的元素。
Object set(int index, Object element)	用指定的元素替换此向量中指定位置处的元素。
void setElementAt(Object obj, int index)	将此向量指定 index 处的组件设置为指定的对象。
void setSize(int newSize)	设置此向量的大小。
int size()	返回此向量中的组件数。
List subList(int fromIndex, int toIndex)	返回此 List 的部分视图，元素范围为从 fromIndex（包括）到 toIndex（不包括）。
Object[] toArray()	返回一个数组，包含此向量中以恰当顺序存放的所有元素。
Object[] toArray(Object[] a)	返回一个数组，包含此向量中以恰当顺序存放的所有元素；返回数组的运行时类型为指定数组的类型。
String toString()	返回此向量的字符串表示形式，其中包含每个元素的 String 表示形式。
void trimToSize()	对此向量的容量进行微调，使其等于向量的当前大小。

实例

下面的程序说明这个集合所支持的几种方法：

```
import java.util.*;

public class VectorDemo {

   public static void main(String args[]) {
      // initial size is 3, increment is 2
      Vector v = new Vector(3, 2);
      System.out.println("Initial size: " + v.size());
      System.out.println("Initial capacity: " +
      v.capacity());
      v.addElement(new Integer(1));
      v.addElement(new Integer(2));
      v.addElement(new Integer(3));
      v.addElement(new Integer(4));
      System.out.println("Capacity after four additions: " +
          v.capacity());

      v.addElement(new Double(5.45));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Double(6.08));
      v.addElement(new Integer(7));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Float(9.4));
      v.addElement(new Integer(10));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Integer(11));
      v.addElement(new Integer(12));
      System.out.println("First element: " +
         (Integer)v.firstElement());
      System.out.println("Last element: " +
         (Integer)v.lastElement());
      if(v.contains(new Integer(3)))
         System.out.println("Vector contains 3.");
      // enumerate the elements in the vector.
      Enumeration vEnum = v.elements();
      System.out.println("\nElements in vector:");
      while(vEnum.hasMoreElements())
         System.out.print(vEnum.nextElement() + " ");
      System.out.println();
   }
}
```

以上实例编译运行结果如下：

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

Java Stack 类

栈是Vector的一个子类，它实现了一个标准的后进先出的栈。

堆栈只定义了默认构造函数，用来创建一个空栈。 堆栈除了包括由Vector定义的所有方法，也定义了自己的一些方法。

```
Stack()
```

除了由Vector定义的所有方法，自己也定义了一些方法：

方法	描述
boolean empty()	测试堆栈是否为空。
Object peek( )	查看堆栈顶部的对象，但不从堆栈中移除它。
Object pop( )	移除堆栈顶部的对象，并作为此函数的值返回该对象。
Object push(Object element)	把项压入堆栈顶部。
int search(Object element)	返回对象在堆栈中的位置，以 1 为基数。

实例

下面的程序说明这个集合所支持的几种方法

```
import java.util.*;

public class StackDemo {

   static void showpush(Stack st, int a) {
      st.push(new Integer(a));
      System.out.println("push(" + a + ")");
      System.out.println("stack: " + st);
   }

   static void showpop(Stack st) {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);
   }

   public static void main(String args[]) {
      Stack st = new Stack();
      System.out.println("stack: " + st);
      showpush(st, 42);
      showpush(st, 66);
      showpush(st, 99);
      showpop(st);
      showpop(st);
      showpop(st);
      try {
         showpop(st);
      } catch (EmptyStackException e) {
         System.out.println("empty stack");
      }
   }
}
```

以上实例编译运行结果如下：

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

Java Dictionary 类

Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似。

给出键和值，你就可以将值存储在Dictionary对象中。一旦该值被存储，就可以通过它的键来获取它。所以和Map一样， Dictionary 也可以作为一个键/值对列表。

Dictionary定义的抽象方法如下表所示：

方法	描述
Enumeration elements( )	返回此 dictionary 中值的枚举。
Object get(Object key)	返回此 dictionary 中该键所映射到的值。
boolean isEmpty( )	测试此 dictionary 是否不存在从键到值的映射。
Enumeration keys( )	返回此 dictionary 中的键的枚举。
Object put(Object key, Object value)	将指定 key 映射到此 dictionary 中指定 value。
Object remove(Object key)	从此 dictionary 中移除 key （及其相应的 value）。
int size( )	返回此 dictionary 中条目（不同键）的数量。

Dictionary类已经过时了。在实际开发中，你可以实现Map接口来获取键/值的存储功能。

Java Hashtable 接口

Hashtable是原始的java.util的一部分， 是一个Dictionary具体的实现 。

然而，Java 2 重构的Hashtable实现了Map接口，因此，Hashtable现在集成到了集合框架中。它和HashMap类很相似，但是它支持同步。

像HashMap一样，Hashtable在哈希表中存储键/值对。当使用一个哈希表，要指定用作键的对象，以及要链接到该键的值。

然后，该键经过哈希处理，所得到的散列码被用作存储在该表中值的索引。

Hashtable定义了四个构造方法。第一个是默认构造方法：

```
Hashtable()
```

第二个构造函数创建指定大小的哈希表：

```
Hashtable(int size)
```

第三个构造方法创建了一个指定大小的哈希表，并且通过fillRatio指定填充比例。

填充比例必须介于0.0和1.0之间，它决定了哈希表在重新调整大小之前的充满程度：

```
Hashtable(int size,float fillRatio)
```

第四个构造方法创建了一个以M中元素为初始化元素的哈希表。

哈希表的容量被设置为M的两倍。

```
Hashtable(Map m)
```

Hashtable中除了从Map接口中定义的方法外，还定义了以下方法：

方法	描述
void clear( )	将此哈希表清空，使其不包含任何键。
Object clone( )	创建此哈希表的浅表副本。
boolean contains(Object value)	测试此映射表中是否存在与指定值关联的键。
boolean containsKey(Object key)	测试指定对象是否为此哈希表中的键。
boolean containsValue(Object value)	如果此 Hashtable 将一个或多个键映射到此值，则返回 true。
Enumeration elements( )	返回此哈希表中的值的枚举。
Object get(Object key)	返回指定键所映射到的值，如果此映射不包含此键的映射，则返回 null. 更确切地讲，如果此映射包含满足 (key.equals(k)) 的从键 k 到值 v 的映射，则此方法返回 v；否则，返回 null。
boolean isEmpty( )	测试此哈希表是否没有键映射到值。
Enumeration keys( )	返回此哈希表中的键的枚举。
Object put(Object key, Object value)	将指定 key 映射到此哈希表中的指定 value。
void rehash( )	增加此哈希表的容量并在内部对其进行重组，以便更有效地容纳和访问其元素。
Object remove(Object key)	从哈希表中移除该键及其相应的值。
int size( )	返回此哈希表中的键的数量。
String toString( )	返回此 Hashtable 对象的字符串表示形式，其形式为 ASCII 字符 ", " （逗号加空格）分隔开的、括在括号中的一组条目。

实例

下面的程序说明这个数据结构支持的几个方法：

```
import java.util.*;

public class HashTableDemo {

   public static void main(String args[]) {
      // Create a hash map
      Hashtable balance = new Hashtable();
      Enumeration names;
      String str;
      double bal;

      balance.put("Zara", new Double(3434.34));
      balance.put("Mahnaz", new Double(123.22));
      balance.put("Ayan", new Double(1378.00));
      balance.put("Daisy", new Double(99.22));
      balance.put("Qadir", new Double(-19.08));

      // Show all balances in hash table.
      names = balance.keys();
      while(names.hasMoreElements()) {
         str = (String) names.nextElement();
         System.out.println(str + ": " +
         balance.get(str));
      }
      System.out.println();
      // Deposit 1,000 into Zara's account
      bal = ((Double)balance.get("Zara")).doubleValue();
      balance.put("Zara", new Double(bal+1000));
      System.out.println("Zara's new balance: " +
      balance.get("Zara"));
   }
}
```

以上实例编译运行结果如下：

```
Qadir: -19.08
Zara: 3434.34
Mahnaz: 123.22
Daisy: 99.22
Ayan: 1378.0

Zara's new balance: 4434.34
```

Java Properties 接口

Properties 继承于 Hashtable.表示一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。

Properties 类被许多Java类使用。例如，在获取环境变量时它就作为System.getProperties()方法的返回值。

Properties 定义如下实例变量.这个变量持有一个Properties对象相关的默认属性列表。

```
Properties defaults;
```

Properties类定义了两个构造方法. 第一个构造方法没有默认值。

```
Properties()
```

第二个构造方法使用propDefault 作为默认值。两种情况下，属性列表都为空：

```
Properties(Properties propDefault)
```

除了从Hashtable中所定义的方法，Properties定义了以下方法：

方法	描述
String getProperty(String key)	用指定的键在此属性列表中搜索属性。
String getProperty(String key, String defaultProperty)	用指定的键在属性列表中搜索属性。
void list(PrintStream streamOut)	将属性列表输出到指定的输出流。
void list(PrintWriter streamOut)	将属性列表输出到指定的输出流。
void load(InputStream streamIn) throws IOException	从输入流中读取属性列表（键和元素对）。
Enumeration propertyNames( )	按简单的面向行的格式从输入字符流中读取属性列表（键和元素对）。
Object setProperty(String key, String value)	调用 Hashtable 的方法 put。
void store(OutputStream streamOut, String description)	以适合使用 load(InputStream)方法加载到 Properties 表中的格式，将此 Properties 表中的属性列表（键和元素对）写入输出流。

实例

下面的程序说明这个数据结构支持的几个方法：

```
import java.util.*;

public class PropDemo {

   public static void main(String args[]) {
      Properties capitals = new Properties();
      Set states;
      String str;

      capitals.put("Illinois", "Springfield");
      capitals.put("Missouri", "Jefferson City");
      capitals.put("Washington", "Olympia");
      capitals.put("California", "Sacramento");
      capitals.put("Indiana", "Indianapolis");

      // Show all states and capitals in hashtable.
      states = capitals.keySet(); // get set-view of keys
      Iterator itr = states.iterator();
      while(itr.hasNext()) {
         str = (String) itr.next();
         System.out.println("The capital of " +
            str + " is " + capitals.getProperty(str) + ".");
      }
      System.out.println();

      // look for state not in list -- specify default
      str = capitals.getProperty("Florida", "Not Found");
      System.out.println("The capital of Florida is "
          + str + ".");
   }
}
```

以上实例编译运行结果如下：

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.
```

Java 集合框架

早在Java 2中之前，Java就提供了特设类。比如：Dictionary, Vector, Stack, 和Properties这些类用来存储和操作对象组。

虽然这些类都非常有用，但是它们缺少一个核心的，统一的主题。由于这个原因，使用Vector类的方式和使用Properties类的方式有着很大不同。

集合框架被设计成要满足以下几个目标。

- 该框架必须是高性能的。基本集合（动态数组，链表，树，哈希表）的实现也必须是高效的。
- 该框架允许不同类型的集合，以类似的方式工作，具有高度的互操作性。
- 对一个集合的扩展和适应必须是简单的。

为此，整个集合框架就围绕一组标准接口而设计。你可以直接使用这些接口的标准实现，诸如： LinkedList, HashSet, 和 TreeSet等,除此之外你也可以通过这些接口实现自己的集合。

集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：

- 接口：是代表集合的抽象数据类型。接口允许集合独立操纵其代表的细节。在面向对象的语言，接口通常形成一个层次。
- 实现（类）：是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构。
- 算法：是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现。

除了集合，该框架也定义了几个Map接口和类。Map里存储的是键/值对。尽管Map不是collections，但是它们完全整合在集合中。

集合接口

集合框架定义了一些接口。本节提供了每个接口的概述：

接口	描述
Collection	接口 允许你使用一组对象，是Collection层次结构的根接口。
List	接口 继承于**Collection**和一个 List实例存储一个有序集合的元素。
Set	继承于 **Collection**，是一个不包含重复元素的集合。
SortedSet	继承于Set保存有序的集合。
Map	将唯一的键映射到值。
Map.Entry	描述在一个Map中的一个元素（键/值对）。是一个Map的内部类。
SortedMap	继承于Map，使Key保持在升序排列。
Enumeration	这是一个传统的接口和定义的方法，通过它可以枚举（一次获得一个）对象集合中的元素。这个传统接口已被迭代器取代。

集合类

Java提供了一套实现了Collection接口的标准集合类。其中一些是具体类，这些类可以直接拿来使用，而另外一些是抽象类，提供了接口的部分实现。

标准集合类汇总于下表：

类	描述
AbstractCollection	实现了大部分的集合接口。
AbstractList	继承于AbstractCollection 并且实现了大部分List接口。
AbstractSequentialList	继承于 AbstractList ，提供了对数据元素的链式访问而不是随机访问。
LinkedList	继承于 AbstractSequentialList，实现了一个链表。
ArrayList	通过继承AbstractList，实现动态数组。
AbstractSet	继承于AbstractCollection 并且实现了大部分Set接口。
HashSet	继承了AbstractSet，并且使用一个哈希表。
LinkedHashSet	具有可预知迭代顺序的 `Set` 接口的哈希表和链接列表实现。
TreeSet	继承于AbstractSet，使用元素的自然顺序对元素进行排序.
AbstractMap	实现了大部分的Map接口。
HashMap	继承了HashMap，并且使用一个哈希表。
TreeMap	继承了AbstractMap，并且使用一颗树。
WeakHashMap	继承AbstractMap类，使用弱密钥的哈希表。
LinkedHashMap	继承于HashMap，使用元素的自然顺序对元素进行排序.
IdentityHashMap	继承AbstractMap类，比较文档时使用引用相等。

在前面的教程中已经讨论通过java.util包中定义的类，如下所示：

类	描述
Vector	Vector类实现了一个动态数组。和ArrayList和相似，但是两者是不同的。
Stack	栈是Vector的一个子类，它实现了一个标准的后进先出的栈。
Dictionary	Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似。
Hashtable	Hashtable是原始的java.util的一部分， 是一个Dictionary具体的实现 。
Properties	Properties 继承于 Hashtable.表示一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。
BitSet	一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。

一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。

集合算法

集合框架定义了几种算法，可用于集合和映射。这些算法被定义为集合类的静态方法。

在尝试比较不兼容的类型时，一些方法能够抛出 ClassCastException异常。当试图修改一个不可修改的集合时，抛出UnsupportedOperationException异常。

集合定义三个静态的变量：EMPTY_SET EMPTY_LIST，EMPTY_MAP的。这些变量都不可改变。

算法描述
Collection Algorithms 这里是一个列表中的所有算法实现。

如何使用迭代器

通常情况下，你会希望遍历一个集合中的元素。例如，显示集合中的每个元素。

做到这一点最简单的方法是采用一个迭代器，它是一个对象，实现了Iterator 接口或ListIterator接口。

迭代器，使你能够通过循环来得到或删除集合的元素。ListIterator继承了Iterator，以允许双向遍历列表和修改元素。

这里通过实例列出Iterator和listIterator接口提供的所有方法。

迭代器方法描述
使用 Java Iterator

如何使用比较器

TreeSet和TreeMap的按照排序顺序来存储元素. 然而，这是通过比较器来精确定义按照什么样的排序顺序。

这个接口可以让我们以不同的方式来排序一个集合。

比较器方法描述
使用 Java Comparator 这里通过实例列出Comparator接口提供的所有方法

总结

Java集合框架为程序员提供了预先包装的数据结构和算法来操纵他们。

集合是一个对象，可容纳其他对象的引用。集合接口声明对每一种类型的集合可以执行的操作。

集合框架的类和接口均在java.util包中。

Java 泛型

如果我们只写一个排序方法，就能够对整形数组、字符串数组甚至支持排序的任何类型的数组进行排序，这该多好啊。

Java泛型方法和泛型类支持程序员使用一个方法指定一组相关方法，或者使用一个类指定一组相关的类型。

Java泛型（generics）是JDK 5中引入的一个新特性,泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。

使用Java泛型的概念，我们可以写一个泛型方法来对一个对象数组排序。然后，调用该泛型方法来对整型数组、浮点数数组、字符串数组等进行排序。

泛型方法

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

下面是定义泛型方法的规则：

- 所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的<E>）。
- 每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。
- 类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。
- 泛型方法方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像int,double,char的等）。

实例

下面的例子演示了如何使用泛型方法打印不同字符串的元素：

```
public class GenericMethodTest
{
   // 泛型方法 printArray
   public static < E > void printArray( E[] inputArray )
   {
      // 输出数组元素
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // 创建不同类型数组： Integer, Double 和 Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // 传递一个整型数组

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // 传递一个双精度型数组

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // 传递一个字符型型数组
    }
}
```

编译以上代码，运行结果如下所示：

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

有界的类型参数:

可能有时候，你会想限制那些被允许传递到一个类型参数的类型种类范围。例如，一个操作数字的方法可能只希望接受Number或者Number子类的实例。这就是有界类型参数的目的。

要声明一个有界的类型参数，首先列出类型参数的名称，后跟extends关键字，最后紧跟它的上界。

实例

下面的例子演示了"extends"如何使用在一般意义上的意思"extends"（类）或者"implements"（接口）。该例子中的泛型方法返回三个可比较对象的最大值。

```
public class MaximumTest
{
   // 比较三个值并返回最大值
   public static <T extends Comparable<T>> T maximum(T x, T y, T z)
   {
      T max = x; // 假设x是初始最大值
      if ( y.compareTo( max ) > 0 ){
         max = y; //y 更大
      }
      if ( z.compareTo( max ) > 0 ){
         max = z; // 现在 z 更大
      }
      return max; // 返回最大对象
   }
   public static void main( String args[] )
   {
      System.out.printf( "Max of %d, %d and %d is %d\n\n",
                   3, 4, 5, maximum( 3, 4, 5 ) );

      System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
                   6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

      System.out.printf( "Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum( "pear", "apple", "orange" ) );
   }
}
```

编译以上代码，运行结果如下所示：

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。

和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

实例

如下实例演示了我们如何定义一个泛型类:

```
public class Box<T> {

  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
     Box<Integer> integerBox = new Box<Integer>();
     Box<String> stringBox = new Box<String>();

     integerBox.add(new Integer(10));
     stringBox.add(new String("Hello World"));

     System.out.printf("Integer Value :%d\n\n", integerBox.get());
     System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

编译以上代码，运行结果如下所示：

```
Integer Value :10

String Value :Hello World
```

Java序列化

Java 提供了一种对象序列化的机制，该机制中，一个对象可以被表示为一个字节序列，该字节序列包括该对象的数据、有关对象的类型的信息和存储在对象中数据的类型。

将序列化对象写入文件之后，可以从文件中读取出来，并且对它进行反序列化，也就是说，对象的类型信息、对象的数据，还有对象中的数据类型可以用来在内存中新建对象。

整个过程都是Java虚拟机（JVM）独立的，也就是说，在一个平台上序列化的对象可以在另一个完全不同的平台上反序列化该对象。

类ObjectInputStream 和ObjectOutputStream是高层次的数据流，它们包含序列化和反序列化对象的方法。

ObjectOutputStream 类包含很多写方法来写各种数据类型，但是一个特别的方法例外：

```
public final void writeObject(Object x) throws IOException
```

上面的方法序列化一个对象，并将它发送到输出流。相似的ObjectInputStream 类包含如下反序列化一个对象的方法：

```
public final Object readObject() throws IOException,
                                 ClassNotFoundException
```

该方法从流中取出下一个对象，并将对象反序列化。它的返回值为Object，因此，你需要将它转换成合适的数据类型。

为了演示序列化在Java中是怎样工作的，我将使用之前教程中提到的Employee类，假设我们定义了如下的Employee类，该类实现了Serializable 接口。

```
public class Employee implements java.io.Serializable
{
   public String name;
   public String address;
   public transient int SSN;
   public int number;
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + name
                           + " " + address);
   }
}
```

请注意，一个类的对象要想序列化成功，必须满足两个条件：

该类必须实现 java.io.Serializable 对象。

该类的所有属性必须是可序列化的。如果有一个属性不是可序列化的，则该属性必须注明是短暂的。

如果你想知道一个Java标准类是否是可序列化的，请查看该类的文档。检验一个类的实例是否能序列化十分简答， 只需要查看该类有没有实现java.io.Serializable接口。

序列化对象

ObjectOutputStream 类用来序列化一个对象，如下的SerializeDemo例子实例化了一个Employee对象，并将该对象序列化到一个文件中。

该程序执行后，就创建了一个名为employee.ser文件。该程序没有任何输出，但是你可以通过代码研读来理解程序的作用。

注意： 当序列化一个对象到文件时， 按照Java的标准约定是给文件一个.ser扩展名。

```
import java.io.*;

public class SerializeDemo
{
   public static void main(String [] args)
   {
      Employee e = new Employee();
      e.name = "Reyan Ali";
      e.address = "Phokka Kuan, Ambehta Peer";
      e.SSN = 11122333;
      e.number = 101;
      try
      {
         FileOutputStream fileOut =
         new FileOutputStream("/tmp/employee.ser");
         ObjectOutputStream out = new ObjectOutputStream(fileOut);
         out.writeObject(e);
         out.close();
         fileOut.close();
         System.out.printf("Serialized data is saved in /tmp/employee.ser");
      }catch(IOException i)
      {
          i.printStackTrace();
      }
   }
}
```

反序列化对象

下面的DeserializeDemo程序反序列化在SerializeDemo程序中创建Employee对象。

```
import java.io.*;
public class DeserializeDemo
{
   public static void main(String [] args)
   {
      Employee e = null;
      try
      {
         FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Employee) in.readObject();
         in.close();
         fileIn.close();
      }catch(IOException i)
      {
         i.printStackTrace();
         return;
      }catch(ClassNotFoundException c)
      {
         System.out.println("Employee class not found");
         c.printStackTrace();
         return;
      }
      System.out.println("Deserialized Employee...");
      System.out.println("Name: " + e.name);
      System.out.println("Address: " + e.address);
      System.out.println("SSN: " + e.SSN);
      System.out.println("Number: " + e.number);
    }
}
```

以上程序编译运行结果如下所示：

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

这里要注意以下要点：

readObject() 方法中的try/catch代码块尝试捕获 ClassNotFoundException异常。对于JVM可以反序列化对象，它必须是能够找到字节码的类。如果JVM在反序列化对象的过程中找不到该类，则抛出一个 ClassNotFoundException异常。

注意，readObject()方法的返回值被转化成Employee引用。

当对象被序列化时，属性SSN的值为111222333，但是因为该属性是短暂的，该值没有被发送到输出流。所以反序列化后Employee对象的SSN属性为0。

Java 网络编程

网络编程是指编写运行在多个设备（计算机）的程序，这些设备都通过网络连接起来。

java.net包中J2SE的API包含有类和接口，它们提供低层次的通信细节。你可以直接使用这些类和接口，来专注于解决问题，而不用关注通信细节。

java.net包中提供了两种常见的网络协议的支持：

- **TCP**： TCP是传输控制协议的缩写，它保障了两个应用程序之间的可靠通信。通常用于互联网协议，被称TCP / IP。
- **UDP**:UDP是用户数据报协议的缩写，一个无连接的协议。提供了应用程序之间要发送的数据的数据包。

本教程主要讲解以下两个主题。

- **Socket** 编程: 这是使用最广泛的网络概念，它已被解释地非常详细
- **URL** 处理: 这部分会在另外的篇幅里讲，点击这里更详细地了解在Java语言中的URL处理。

Socket 编程

套接字使用TCP提供了两台计算机之间的通信机制。 客户端程序创建一个套接字，并尝试连接服务器的套接字。

当连接建立时，服务器会创建一个Socket对象。客户端和服务器现在可以通过对Socket对象的写入和读取来进行进行通信。

java.net.Socket类代表一个套接字，并且java.net.ServerSocket类为服务器程序提供了一种来监听客户端，并与他们建立连接的机制。

以下步骤在两台计算机之间使用套接字建立TCP连接时会出现：

- 服务器实例化一个ServerSocket对象，表示通过服务器上的端口通信。
- 服务器调用 ServerSocket类 的accept（）方法，该方法将一直等待，直到客户端连接到服务器上给定的端口。
- 服务器正在等待时，一个客户端实例化一个Socket对象，指定服务器名称和端口号来请求连接。
- Socket类的构造函数试图将客户端连接到指定的服务器和端口号。如果通信被建立，则在客户端创建一个Socket对象能够与服务器进行通信。
- 在服务器端，accept()方法返回服务器上一个新的socket引用，该socket连接到客户端的socket。

连接建立后，通过使用I/O流在进行通信。每一个socket都有一个输出流和一个输入流。客户端的输出流连接到服务器端的输入流，而客户端的输入流连接到服务器端的输出流。

TCP是一个双向的通信协议，因此数据可以通过两个数据流在同一时间发送.以下是一些类提供的一套完整的有用的方法来实现sockets。

ServerSocket 类的方法

服务器应用程序通过使用java.net.ServerSocket类以获取一个端口,并且侦听客户端请求。

ServerSocket类有四个构造方法：

方法	描述
public ServerSocket(int port) throws IOException	创建绑定到特定端口的服务器套接字。
public ServerSocket(int port, int backlog) throws IOException	利用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口号。
public ServerSocket(int port, int backlog, InetAddress address) throws IOException	使用指定的端口、侦听 backlog 和要绑定到的本地 IP 地址创建服务器。
public ServerSocket() throws IOException	创建非绑定服务器套接字。

创建非绑定服务器套接字。 如果ServerSocket构造方法没有抛出异常，就意味着你的应用程序已经成功绑定到指定的端口，并且侦听客户端请求。

这里有一些ServerSocket类的常用方法：

方法	描述
public int getLocalPort()	返回此套接字在其上侦听的端口。
public Socket accept() throws IOException	侦听并接受到此套接字的连接。
public void setSoTimeout(int timeout)	通过指定超时值启用/禁用 SO_TIMEOUT，以毫秒为单位。
public void bind(SocketAddress host, int backlog)	将 ServerSocket 绑定到特定地址（IP 地址和端口号）。

Socket 类的方法

java.net.Socket类代表客户端和服务器都用来互相沟通的套接字。客户端要获取一个Socket对象通过实例化 ， 而 服务器获得一个Socket对象则通过accept()方法的返回值。

Socket类有五个构造方法.

方法	描述
public Socket(String host, int port) throws UnknownHostException, IOException.	创建一个流套接字并将其连接到指定主机上的指定端口号。
public Socket(InetAddress host, int port) throws IOException	创建一个流套接字并将其连接到指定 IP 地址的指定端口号。
public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.	创建一个套接字并将其连接到指定远程主机上的指定远程端口。
public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.	创建一个套接字并将其连接到指定远程地址上的指定远程端口。
public Socket()	通过系统默认类型的 SocketImpl 创建未连接套接字

当Socket构造方法返回，并没有简单的实例化了一个Socket对象，它实际上会尝试连接到指定的服务器和端口。

下面列出了一些感兴趣的方法，注意客户端和服务器端都有一个Socket对象，所以无论客户端还是服务端都能够调用这些方法。

方法	描述
public void connect(SocketAddress host, int timeout) throws IOException	将此套接字连接到服务器，并指定一个超时值。
public InetAddress getInetAddress()	返回套接字连接的地址。
public int getPort()	返回此套接字连接到的远程端口。
public int getLocalPort()	返回此套接字绑定到的本地端口。
public SocketAddress getRemoteSocketAddress()	返回此套接字连接的端点的地址，如果未连接则返回 null。
public InputStream getInputStream() throws IOException	返回此套接字的输入流。
public OutputStream getOutputStream() throws IOException	返回此套接字的输出流。
public void close() throws IOException	关闭此套接字。

InetAddress 类的方法

这个类表示互联网协议(IP)地址。下面列出了Socket编程时比较有用的方法：

方法	描述
static InetAddress getByAddress(byte[] addr)	在给定原始 IP 地址的情况下，返回 InetAddress 对象。
static InetAddress getByAddress(String host, byte[] addr)	根据提供的主机名和 IP 地址创建 InetAddress。
static InetAddress getByName(String host)	在给定主机名的情况下确定主机的 IP 地址。
String getHostAddress()	返回 IP 地址字符串（以文本表现形式）。
String getHostName()	获取此 IP 地址的主机名。
static InetAddress getLocalHost()	返回本地主机。
String toString()	将此 IP 地址转换为 String。

Socket 客户端实例

如下的GreetingClient 是一个客户端程序，改程序通过socket连接到服务器并发送一个问候，然后等待一个响应。

```
// 文件名 GreetingClient.java
<pre>
import java.net.*;
import java.io.*;

public class GreetingClient
{
   public static void main(String [] args)
   {
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try
      {
         System.out.println("Connecting to " + serverName
                             + " on port " + port);
         Socket client = new Socket(serverName, port);
         System.out.println("Just connected to "
                      + client.getRemoteSocketAddress());
         OutputStream outToServer = client.getOutputStream();
         DataOutputStream out =
                       new DataOutputStream(outToServer);

         out.writeUTF("Hello from "
                      + client.getLocalSocketAddress());
         InputStream inFromServer = client.getInputStream();
         DataInputStream in =
                        new DataInputStream(inFromServer);
         System.out.println("Server says " + in.readUTF());
         client.close();
      }catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

Socket 服务器实例

如下的GreetingServer 程序是一个服务器端应用程序，改程序使用Socket来监听一个指定的端口。

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

像下面一样开启客户端：

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

Java 发送邮件

使用Java应用程序发送E-mail十分简单，但是首先你应该在你的机器上安装JavaMail API 和 Java Activation Framework (JAF) 。

你可以在 JavaMail (Version 1.2) 下载最新的版本。

你可以再 在JAF (Version 1.1.1)下载最新的版本。

下载并解压这些文件，最上层文件夹你会发现很多的jar文件。你需要将mail.jar和 activation.jar 添加到你的CLASSPATH中。

如果你使用第三方邮件服务器如QQ的SMTP服务器，可查看文章底部用户认证完整的实例。

发送一封简单的 E-mail

下面是一个发送简单E-mail的例子。假设你的localhost已经连接到网络。

```
// 文件名 SendEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail
{
   public static void main(String [] args)
   {
      // 收件人电子邮箱
      String to = "abcd@gmail.com";

      // 发件人电子邮箱
      String from = "web@gmail.com";

      // 指定发送邮件的主机为 localhost
      String host = "localhost";

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认session对象
      Session session = Session.getDefaultInstance(properties);

      try{
         // 创建默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // Set Subject: 头部头字段
         message.setSubject("This is the Subject Line!");

         // 设置消息体
         message.setText("This is actual message");

         // 发送消息
         Transport.send(message);
         System.out.println("Sent message successfully....");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

编译并运行这个程序来发送一封简单的E-mail：

```
$ java SendEmail
Sent message successfully....
```

如果你想发送一封e-mail给多个收件人，那么使用下面的方法来指定多个收件人ID：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

下面是对于参数的描述：

- **type:**要被设置为TO, CC 或者BCC. 这里CC 代表抄送、BCC 代表秘密抄送y. 举例：*Message.RecipientType.TO*
- **addresses:** 这是email ID的数组。在指定电子邮件ID时，你将需要使用InternetAddress()方法。

发送一封HTML E-mail

下面是一个发送HTML E-mail的例子。假设你的localhost已经连接到网络。

和上一个例子很相似，除了我们要使用setContent()方法来通过第二个参数为"text/html"，来设置内容来指定要发送HTML内容。

```
// 文件名 SendHTMLEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendHTMLEmail
{
   public static void main(String [] args)
   {

      // 收件人电子邮箱
      String to = "abcd@gmail.com";

      // 发件人电子邮箱
      String from = "web@gmail.com";

      // 指定发送邮件的主机为 localhost
      String host = "localhost";

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象。
      Session session = Session.getDefaultInstance(properties);

      try{
         // 创建默认的 MimeMessage 对象。
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // Set Subject: 头字段
         message.setSubject("This is the Subject Line!");

         // 发送 HTML 消息, 可以插入html标签
         message.setContent("<h1>This is actual message</h1>",
                            "text/html" );

         // 发送消息
         Transport.send(message);
         System.out.println("Sent message successfully....");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

编译并运行此程序来发送HTML e-mail：

```
$ java SendHTMLEmail
Sent message successfully....
```

发送带有附件的 E-mail

下面是一个发送带有附件的 E-mail的例子。假设你的localhost已经连接到网络。

```
// 文件名 SendFileEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendFileEmail
{
   public static void main(String [] args)
   {

      // 收件人电子邮箱
      String to = "abcd@gmail.com";

      // 发件人电子邮箱
      String from = "web@gmail.com";

      // 指定发送邮件的主机为 localhost
      String host = "localhost";

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象。
      Session session = Session.getDefaultInstance(properties);

      try{
         // 创建默认的 MimeMessage 对象。
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // Set Subject: 头字段
         message.setSubject("This is the Subject Line!");

         // 创建消息部分
         BodyPart messageBodyPart = new MimeBodyPart();

         // 消息
         messageBodyPart.setText("This is message body");

         // 创建多重消息
         Multipart multipart = new MimeMultipart();

         // 设置文本消息部分
         multipart.addBodyPart(messageBodyPart);

         // 附件部分
         messageBodyPart = new MimeBodyPart();
         String filename = "file.txt";
         DataSource source = new FileDataSource(filename);
         messageBodyPart.setDataHandler(new DataHandler(source));
         messageBodyPart.setFileName(filename);
         multipart.addBodyPart(messageBodyPart);

         // 发送完整消息
         message.setContent(multipart );

         //   发送消息
```

```
            Transport.send(message);
            System.out.println("Sent message successfully....");
         }catch (MessagingException mex) {
            mex.printStackTrace();
         }
      }
   }
```

编译并运行你的程序来发送一封带有附件的邮件。

```
$ java SendFileEmail
Sent message successfully....
```

用户认证部分

如果需要提供用户名和密码给e-mail服务器来达到用户认证的目的，你可以通过如下设置来完成：

```
 props.put("mail.smtp.auth", "true");
 props.setProperty("mail.user", "myuser");
 props.setProperty("mail.password", "mypwd");
```

e-mail其他的发送机制和上述保持一致。

需要用户名密码验证邮件发送实例:

本实例以QQ邮件服务器为例，你需要在登录QQ邮箱后台在"设置"=》账号中开启POP3/SMTP服务 ，如下图所示：

Java 代码如下：

```
// 需要用户名密码邮件发送实例
//文件名 SendEmail2.java
//本实例以QQ邮箱为例，你需要在qq后台设置

import java.util.Properties;

import javax.mail.Authenticator;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendEmail2
{
   public static void main(String [] args)
   {
      // 收件人电子邮箱
      String to = "xxx@qq.com";

      // 发件人电子邮箱
      String from = "xxx@qq.com";

      // 指定发送邮件的主机为 localhost
      String host = "smtp.qq.com";  //QQ 邮件服务器

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      properties.put("mail.smtp.auth", "true");
      // 获取默认session对象
      Session session = Session.getDefaultInstance(properties,new Authenticator(){
        public PasswordAuthentication getPasswordAuthentication()
        {
         return new PasswordAuthentication("xxx@qq.com", "qq邮箱密码"); //发件人邮件用户名、密
        }
       });

      try{
         // 创建默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                                  new InternetAddress(to));

         // Set Subject: 头部头字段
         message.setSubject("This is the Subject Line!");

         // 设置消息体
         message.setText("This is actual message");

         // 发送消息
         Transport.send(message);
         System.out.println("Sent message successfully....from w3cschool.cc");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

Java 多线程编程

Java给多线程编程提供了内置的支持。一个多线程程序包含两个或多个能并发运行的部分。程序的每一部分都称作一个线程，并且每个线程定义了一个独立的执行路径。

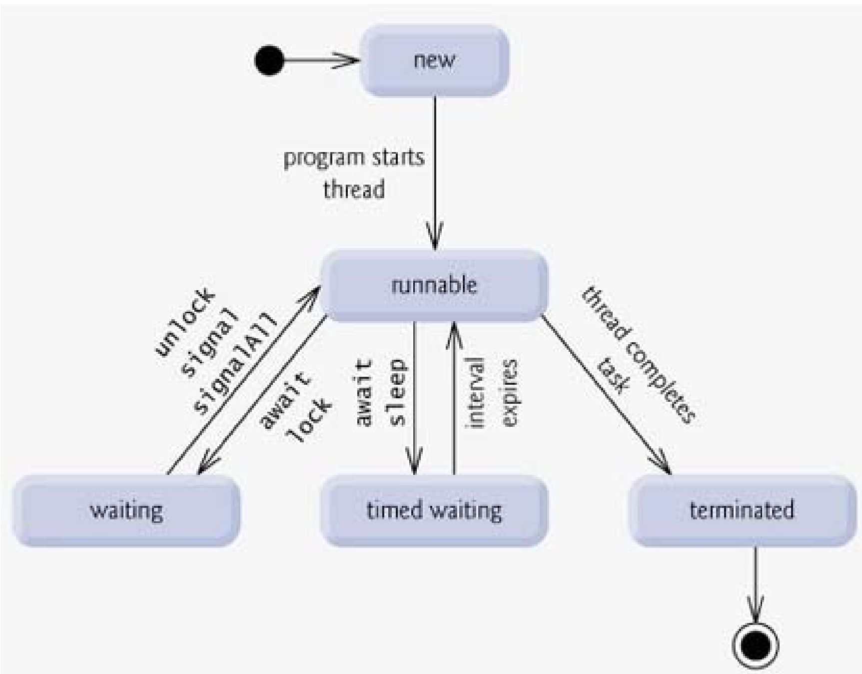
多线程是多任务的一种特别的形式。多线程比多任务需要更小的开销。

这里定义和线程相关的另一个术语：进程：一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的非守候线程都结束运行后才能结束。

多线程能满足程序员编写非常有效率的程序来达到充分利用CPU的目的，因为CPU的空闲时间能够保持在最低限度。

一个线程的生命周

线程经过其生命周期的各个阶段。下图显示了一个线程完整的生命周期。



- 新状态: 一个新产生的线程从新状态开始了它的生命周期。它保持这个状态知道程序start这个线程。
- 运行状态:当一个新状态的线程被start以后，线程就变成可运行状态，一个线程在此状态下被认为是开始执行其任务
- 就绪状态:当一个线程等待另外一个线程执行一个任务的时候，该线程就进入就绪状态。当另一个线程给就绪状态的线程发送信号时，该线程才重新切换到运行状态。
- 休眠状态: 由于一个线程的时间片用完了，该线程从运行状态进入休眠状态。当时间间隔

到期或者等待的时间发生了，该状态的线程切换到运行状态。
- 终止状态: 一个运行状态的线程完成任务或者其他终止条件发生，该线程就切换到终止状态。

线程的优先级

每一个Java线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。Java优先级在MIN_PRIORITY（1）和MAX_PRIORITY（10）之间的范围内。默认情况下，每一个线程都会分配一个优先级NORM_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器时间。然而，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

创建一个线程

Java提供了两种创建线程方法：

- 通过实现Runable接口；
- 通过继承Thread类本身。

通过实现Runnable接口来创建线程

创建一个线程，最简单的方法是创建一个实现Runnable接口的类。

为了实现Runnable，一个类只需要执行一个方法调用run()，声明如下：

```
publicvoid run()
```

你可以重写该方法，重要的是理解的run()可以调用其他方法，使用其他类，并声明变量，就像主线程一样。

在创建一个实现Runnable接口的类之后，你可以在类中实例化一个线程对象。

Thread定义了几个构造方法，下面的这个是我们经常使用的：

```
Thread(Runnable threadOb,String threadName);
```

这里，threadOb 是一个实现Runnable 接口的类的实例，并且 threadName指定新线程的名字。

新线程创建之后，你调用它的start()方法它才会运行。

```
void start();
```

实例

下面是一个创建线程并开始让它执行的实例：

```
// 创建一个新的线程
class NewThread implements Runnable {
   Thread t;
   NewThread() {
      // 创建第二个新线程
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread: " + t);
      t.start(); // 开始线程
   }

   // 第二个线程入口
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            // 暂停线程
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
     }
     System.out.println("Exiting child thread.");
   }
}

public class ThreadDemo {
   public static void main(String args[]) {
      new NewThread(); // 创建一个新线程
      try {
         for(int i = 5; i > 0; i--) {
           System.out.println("Main Thread: " + i);
           Thread.sleep(100);
         }
      } catch (InterruptedException e) {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

编译以上程序运行结果如下：

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

通过继承Thread来创建线程

创建一个线程的第二种方法是创建一个新的类，该类继承Thread类，然后创建一个该类的实例。

继承类必须重写run()方法，该方法是新线程的入口点。它也必须调用start()方法才能执行。

实例

```
// 通过继承 Thread 创建线程
class NewThread extends Thread {
   NewThread() {
      // 创建第二个新线程
      super("Demo Thread");
      System.out.println("Child thread: " + this);
      start(); // 开始线程
   }

   // 第二个线程入口
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
                            // 让线程休眠一会
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
      }
      System.out.println("Exiting child thread.");
   }
}

public class ExtendThread {
   public static void main(String args[]) {
      new NewThread(); // 创建一个新线程
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Main Thread: " + i);
            Thread.sleep(100);
         }
      } catch (InterruptedException e) {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

编译以上程序运行结果如下：

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Thread 方法

下表列出了Thread类的一些重要方法：

方法	描述	
public void start()	使该线程开始执行；**Java**	虚拟机调用该线程的 run 方法。
public void run()	如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。	
public final void setName(String name)	改变线程名称，使之与参数 name 相同。	
public final void setPriority(int priority)	更改线程的优先级。	
public final void setDaemon(boolean on)	将该线程标记为守护线程或用户线程。	
public final void join(long millisec)	等待该线程终止的时间最长为 millis 毫秒。	
public void interrupt()	中断线程。	
public final boolean isAlive()	测试线程是否处于活动状态。	

测试线程是否处于活动状态。 上述方法是被Thread对象调用的。下面的方法是Thread类的静态方法。

方法	描述
public static void yield()	暂停当前正在执行的线程对象，并执行其他线程。
public static void sleep(long millisec)	在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。
public static boolean holdsLock(Object x)	当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。
public static Thread currentThread()	返回对当前正在执行的线程对象的引用。
public static void dumpStack()	将当前线程的堆栈跟踪打印至标准错误流。

实例

如下的ThreadClassDemo 程序演示了Thread类的一些方法：

```
// 文件名 : DisplayMessage.java
// 通过实现 Runnable 接口创建线程
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}
```

```
// 文件名 : GuessANumber.java
// 通过继承 Thread 类创建线程

public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
          guess = (int) (Math.random() * 100 + 1);
          System.out.println(this.getName()
                       + " guesses " + guess);
          counter++;
      }while(guess != number);
      System.out.println("** Correct! " + this.getName()
                       + " in " + counter + " guesses.**");
   }
}
```

```
// 文件名 : ThreadClassDemo.java
public class ThreadClassDemo
{
   public static void main(String [] args)
   {
      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(hello);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
      System.out.println("Starting goodbye thread...");
      thread2.start();

      System.out.println("Starting thread3...");
      Thread thread3 = new GuessANumber(27);
      thread3.start();
      try
      {
         thread3.join();
      }catch(InterruptedException e)
      {
         System.out.println("Thread interrupted.");
      }
      System.out.println("Starting thread4...");
      Thread thread4 = new GuessANumber(75);

           thread4.start();
      System.out.println("main() is ending...");
   }
}
```

运行结果如下，每一次运行的结果都不一样。

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Thread-2 guesses 27
Hello
** Correct! Thread-2 in 102 guesses.**
Hello
Starting thread4...
Hello
Hello
..........remaining result produced.
```

线程的几个主要概念:

在多线程编程时，你需要了解以下几个概念：

- 线程同步
- 线程间通信
- 线程死锁
- 线程控制：挂起、停止和恢复

多线程的使用

有效利用多线程的关键是理解程序是并发执行而不是串行执行的。例如：程序中有两个子系统需要并发执行，这时候就需要利用多线程编程。

通过对多线程的使用，可以编写出非常高效的程序。不过请注意，如果你创建太多的线程，程序执行的效率实际上是降低了，而不是提升了。

请记住，上下文的切换开销也很重要，如果你创建了太多的线程，CPU花费在上下文的切换的时间将多于执行程序的时间！

Java Applet基础

applet是一种Java程序。它一般运行在支持Java的Web浏览器内。因为它有完整的Java API支持,所以applet是一个全功能的Java应用程序。

如下所示是独立的Java应用程序和applet程序之间重要的不同：

- Java中applet类继承了 java.applet.Applet类
- Applet类没有定义main()，所以一个 Applet程序不会调用main()方法，
- Applets被设计为嵌入在一个HTML页面。
- 当用户浏览包含Applet的HTML页面，Applet的代码就被下载到用户的机器上。
- 要查看一个applet需要JVM。 JVM可以是Web浏览器的一个插件，或一个独立的运行时环境。
- 用户机器上的JVM创建一个applet类的实例，并调用Applet生命周期过程中的各种方法。
- Applets有Web浏览器强制执行的严格的安全规则，applet的安全机制被称为沙箱安全。
- applet需要的其他类可以用Java归档（JAR）文件的形式下载下来。

Applet的生命周期

Applet类中的四个方法给你提供了一个框架，你可以再该框架上开发小程序：

- **init:** 该方法的目的是为你的applet提供所需的任何初始化。在Applet标记内的param标签被处理后调用该方法。
- **start:** 浏览器调用init方法后，该方法被自动调用。每当用户从其他页面返回到包含Applet的页面时，则调用该方法。
- **stop:**当用户从包含applet的页面移除的时候，该方法自动被调用。因此，可以在相同的applet中反复调用该方法。
- **destroy:** 此方法仅当浏览器正常关闭时调用。因为applets只有在HTML网页上有效，所以你不应该在用户离开包含Applet的页面后遗漏任何资源.
- **paint:** 该方法在start()方法之后立即被调用，或者在applet需要重绘在浏览器的时候调用。paint()方法实际上继承于java.awt。

"Hello, World" Applet:

下面是一个简单的Applet程序HelloWorldApplet.java:

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet
{
   public void paint (Graphics g)
   {
      g.drawString ("Hello World", 25, 50);
   }
}
```

这些import语句将以下类导入到我们的applet类中：

```
java.applet.Applet.
java.awt.Graphics.
```

没有这些import语句，Java编译器就识别不了Applet和Graphics类。

Applet 类

每一个applet都是java.applet.Applet 类的子类，基础的Applet类提供了供衍生类调用的方法，以此来得到浏览器上下文的信息和服务。

这些方法做了如下事情：

- 得到applet的参数
- 得到包含applet的HTML文件的网络位置
- 得到applet类目录的网络位置
- 打印浏览器的状态信息
- 获取一张图片
- 获取一个音频片段
- 播放一个音频片段
- 调整此 applet 的大小

除此之外，Applet类还提供了一个接口，该接口供Viewer或浏览器来获取applet的信息，并且来控制applet的执行。

Viewer可能是：

- 请求applet作者、版本和版权的信息
- 请求applet识别的参数的描述
- 初始化applet
- 销毁applet
- 开始执行applet
- 结束执行applet

Applet类提供了对这些方法的默认实现，这些方法可以在需要的时候重写。

"Hello，World"applet都是按标准编写的。唯一被重写的方法是paint方法。

Applet的调用

applet是一种Java程序。它一般运行在支持Java的Web浏览器内。因为它有完整的Java API支持,所以applet是一个全功能的Java应用程序。

<applet>标签是在HTML文件中嵌入applet的基础。以下是一个调用"Hello World"applet的例子；

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="HelloWorldApplet.class" width="320" height="120">
If your browser was Java-enabled, a "Hello, World"
message would appear here.
</applet>
<hr>
</html>
```

注意: 你可以参照HTML Applet标签来更多的了解从HTML中调用applet的方法。

<applet>标签的属性指定了要运行的Applet类。Width和height用来指定applet运行面板的初始大小。applet必须使用</applet>标签来关闭。

如果applet接受参数，那么参数的值需要在<param>标签里添加，该标签位于<applet>和</applet>之间。浏览器忽略了applet标签之间的文本和其他标签。

不支持Java的浏览器不能执行<applet>和</applet>。因此，在标签之间显示并且和applet没有关系的任何东西，在不支持的Java的浏览器里是可见的。

Viewer或者浏览器在文档的位置寻找编译过的Java代码，要指定文档的路径，得使用<applet>标签的codebase属性指定。

如下所示：

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class" width="320" height="120">
```

如果applet所在一个包中而不是默认包，那么所在的包必须在code属性里指定，例如：

```
<applet code="mypackage.subpackage.TestApplet.class"
           width="320" height="120">
```

获得applet参数

下面的例子演示了如何使用一个applet响应来设置文件中指定的参数。该Applet显示了一个黑色棋盘图案和第二种颜色。

第二种颜色和每一列的大小通过文档中的applet的参数指定。

CheckerApplet 在init()方法里得到它的参数。也可以在paint()方法里得到它的参数。然而，在applet开始得到值并保存了设置，而不是每一次刷新的时候都得到值，这样是很方便，并且高效的。

applet viewer或者浏览器在applet每次运行的时候调用init()方法。在加载applet之后，Viewer立即调用init()方法（Applet.init()什么也没做），重写该方法的默认实现，添加一些自定义的初始化代码。

Applet.getParameter()方法通过给出参数名称得到参数值。如果得到的值是数字或者其他非字符数据，那么必须解析为字符串类型。

下例是CheckerApplet.java的梗概：

```
import java.applet.*;
import java.awt.*;
public class CheckerApplet extends Applet
{
   int squareSize = 50;// 初始化默认大小
   public void init () {}
   private void parseSquareSize (String param) {}
   private Color parseColor (String param) {}
   public void paint (Graphics g) {}
}
```

下面是CheckerApplet类的init()方法和私有的parseSquareSize()方法：

```
public void init ()
{
   String squareSizeParam = getParameter ("squareSize");
   parseSquareSize (squareSizeParam);
   String colorParam = getParameter ("color");
   Color fg = parseColor (colorParam);
   setBackground (Color.black);
   setForeground (fg);
}
private void parseSquareSize (String param)
{
   if (param == null) return;
   try {
      squareSize = Integer.parseInt (param);
   }
   catch (Exception e) {
     // 保留默认值
   }
}
```

该applet调用parseSquareSize()，来解析squareSize参数。parseSquareSize()调用了库方法Integer. parseInt()，该方法将一个字符串解析为一个整数，当参数无效的时候，Integer.parseInt()抛出异常。

因此，parseSquareSize()方法也是捕获异常的，并不允许applet接受无效的输入。

Applet调用parseColor()方法将颜色参数解析为一个Color值。parseColor()方法做了一系列字符串的比较，来匹配参数的值和预定义颜色的名字。你需要实现这些方法来使applet工作。

指定applet参数

如下的例子是一个HTML文件，其中嵌入了CheckerApplet类。HTML文件通过使用<param>标签的方法给applet指定了两个参数。

```
<html>
<title>Checkerboard Applet</title>
<hr>
<applet code="CheckerApplet.class" width="480" height="320">
<param name="color" value="blue">
<param name="squaresize" value="30">
</applet>
<hr>
</html>
```

注意: 参数名字大小写不敏感。

应用程序转换成Applet

将图形化的Java应用程序（是指，使用AWT的应用程序和使用java程序启动器启动的程序）转换成嵌入在web页面里的applet是很简单的。

下面是将应用程序转换成applet的几个步骤：

- 编写一个HTML页面，该页面带有能加载applet代码的标签。
- 编写一个JApplet类的子类，将该类设置为public。否则，applet不能被加载。
- 消除应用程序的main()方法。不要为应用程序构造框架窗口，因为你的应用程序要显示在浏览器中。
- 将应用程序中框架窗口的构造方法里的初始化代码移到applet的init()方法中，你不必显示的构造applet对象，浏览器将通过调用init()方法来实例化一个对象。
- 移除对setSize()方法的调用，对于applet来讲，大小已经通过HTML文件里的width和height参数设定好了。
- 移除对 setDefaultCloseOperation()方法的调用。Applet不能被关闭，它随着浏览器的退出而终止。
- 如果应用程序调用了setTitle()方法，消除对该方法的调用。applet不能有标题栏。（当然你可以给通过html的title标签给网页自身命名）
- 不要调用setVisible(true),applet是自动显示的。

事件处理

Applet类从Container类继承了许多事件处理方法。Container类定义了几个方法，例如：processKeyEvent()和processMouseEvent()，用来处理特别类型的事件，还有一个捕获所有事件的方法叫做processEvent。

为了响应一个事件，applet必须重写合适的事件处理方法。

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet
                             implements MouseListener {

    StringBuffer strBuffer;

    public void init() {
         addMouseListener(this);
         strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }

    public void start() {
        addItem("starting the applet ");
    }

    public void stop() {
        addItem("stopping the applet ");
    }

    public void destroy() {
        addItem("unloading the applet");
    }

    void addItem(String word) {
        System.out.println(word);
        strBuffer.append(word);
        repaint();
    }

    public void paint(Graphics g) {
         //Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
                      getWidth() - 1,
                      getHeight() - 1);

         //display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20);
    }

    public void mouseEntered(MouseEvent event) {
    }
    public void mouseExited(MouseEvent event) {
    }
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }

    public void mouseClicked(MouseEvent event) {
         addItem("mouse clicked! ");
    }
}
```

如下调用该applet：

```
<html>
<title>Event Handling</title>
<hr>
<applet code="ExampleEventHandling.class"
width="300" height="300">
</applet>
<hr>
</html>
```

最开始运行，applet显示 "initializing the applet. Starting the applet."，然后你一点击矩形框，就会显示"mouse clicked" 。

显示图片

applet能显示GIF,JPEG,BMP等其他格式的图片。为了在applet中显示图片，你需要使用java.awt.Graphics类的drawImage()方法。

如下实例演示了显示图片的所有步骤：

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class ImageDemo extends Applet
{
  private Image image;
  private AppletContext context;
  public void init()
  {
      context = this.getAppletContext();
      String imageURL = this.getParameter("image");
      if(imageURL == null)
      {
         imageURL = "java.jpg";
      }
      try
      {
         URL url = new URL(this.getDocumentBase(), imageURL);
         image = context.getImage(url);
      }catch(MalformedURLException e)
      {
         e.printStackTrace();
         // Display in browser status bar
         context.showStatus("Could not load image!");
      }
   }
   public void paint(Graphics g)
   {
      context.showStatus("Displaying image");
      g.drawImage(image, 0, 0, 200, 84, null);
      g.drawString("www.javalicense.com", 35, 100);
   }
}
```

如下调用该applet：

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="300" height="200">
<param name="image" value="java.jpg">
</applet>
<hr>
</html>
```

播放音频

Applet能通过使用java.applet包中的AudioClip接口播放音频。AudioClip接口定义了三个方法：

- **public void play():** 从一开始播放音频片段一次。
- **public void loop():** 循环播放音频片段
- **public void stop():** 停止播放音频片段

为了得到AudioClip对象，你必须调用Applet类的getAudioClip()方法。无论URL指向的是否是一个真实的音频文件，该方法都会立即返回结果。

直到要播放音频文件时，该文件才会下载下来。

如下实例演示了播放音频的所有步骤：

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet
{
   private AudioClip clip;
   private AppletContext context;
   public void init()
   {
      context = this.getAppletContext();
      String audioURL = this.getParameter("audio");
      if(audioURL == null)
      {
         audioURL = "default.au";
      }
      try
      {
         URL url = new URL(this.getDocumentBase(), audioURL);
         clip = context.getAudioClip(url);
      }catch(MalformedURLException e)
      {
         e.printStackTrace();
         context.showStatus("Could not load audio file!");
      }
   }
   public void start()
   {
      if(clip != null)
      {
         clip.loop();
      }
   }
   public void stop()
   {
      if(clip != null)
      {
         clip.stop();
      }
   }
}
```

如下调用applet：

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="0" height="0">
<param name="audio" value="test.wav">
</applet>
<hr>
```

你可以使用你电脑上的test.wav来测试上面的实例。

Java 文档注释

Java只是三种注释方式。前两种分别是// 和/* */，第三种被称作说明注释，它以/** 开始，以 */结束。

说明注释允许你在程序中嵌入关于程序的信息。你可以使用javadoc工具软件来生成信息，并输出到HTML文件中。

说明注释，是你更加方面的记录你的程序的信息。

javadoc 标签

javadoc工具软件识别以下标签：

标签	描述	示例
@author	标识一个类的作者	@author description
@deprecated	指名一个过期的类或成员	@deprecated description
{@docRoot}	指明当前文档根目录的路径	Directory Path
@exception	标志一个类抛出的异常	@exception exception-name explanation
{@inheritDoc}	从直接父类继承的注释	Inherits a comment from the immediate surperclass.
{@link}	插入一个到另一个主题的链接	{@link name text}
{@linkplain}	插入一个到另一个主题的链接，但是该链接显示纯文本字体	Inserts an in-line link to another topic.
@param	说明一个方法的参数	@param parameter-name explanation
@return	说明返回值类型	@return explanation
@see	指定一个到另一个主题的链接	@see anchor
@serial	说明一个序列化属性	@serial description
@serialData	说明通过writeObject( ) 和 writeExternal( )方法写的数据	@serialData description
@serialField	说明一个ObjectStreamField组件	@serialField name type description
@since	标记当引入一个特定的变化时	@since release
@throws	和 @exception标签一样.	The @throws tag has the same meaning as the @exception tag.
{@value}	显示常量的值，该常量必须是static属性。	Displays the value of a constant, which must be a static field.
@version	指定类的版本	@version info

文档注释

在开始的/**之后，第一行或几行是关于类、变量和方法的主要描述.

之后，你可以包含一个或多个何种各样的@标签。每一个@标签必须在一个新行的开始或者在一行的开始紧跟星号(*).

多个相同类型的标签应该放成一组。例如，如果你有三个@see标签，可以将它们一个接一个的放在一起。

下面是一个类的说明注释的示例：

```
/*** This class draws a bar chart.
* @author Zara Ali
* @version 1.2
*/
```

javadoc输出什么

javadoc工具将你Java程序的源代码作为输入，输出一些包含你程序注释的HTML文件。

每一个类的信息将在独自的HTML文件里。javadoc也可以输出继承的树形结构和索引。

由于javadoc的实现不同，工作也可能不同，你需要检查你的Java开发系统的版本等细节，选择合适的Javadoc版本。

实例

下面是一个使用说明注释的简单实例。注意每一个注释都在它描述的项目的前面。

在经过javadoc处理之后，SquareNum类的注释将在SquareNum.html中找到。

```
import java.io.*;

/**
* This class demonstrates documentation comments.
* @author Ayan Amhed
* @version 1.2
*/
public class SquareNum {
   /**
   * This method returns the square of num.
   * This is a multiline description. You can use
   * as many lines as you like.
   * @param num The value to be squared.
   * @return num squared.
   */
   public double square(double num) {
      return num * num;
   }
   /**
   * This method inputs a number from the user.
   * @return The value input as a double.
   * @exception IOException On input error.
   * @see IOException
   */
   public double getNumber() throws IOException {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader inData = new BufferedReader(isr);
      String str;
      str = inData.readLine();
      return (new Double(str)).doubleValue();
   }
   /**
   * This method demonstrates square().
   * @param args Unused.
   * @return Nothing.
   * @exception IOException On input error.
   * @see IOException
   */
   public static void main(String args[]) throws IOException
   {
      SquareNum ob = new SquareNum();
      double val;
      System.out.println("Enter value to be squared: ");
      val = ob.getNumber();
      val = ob.square(val);
      System.out.println("Squared value is " + val);
   }
}
```

如下，使用javadoc工具处理SquareNum.java文件：

```
$ javadoc SquareNum.java
Loading source file SquareNum.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0_13
Building tree for all the packages and classes...
Generating SquareNum.html...
SquareNum.java:39: warning - @return tag cannot be used\
                      in method with void return type.
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...
1 warning
$
```

Servlet 教程

Servlet 简介

Servlet 是什么？

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。

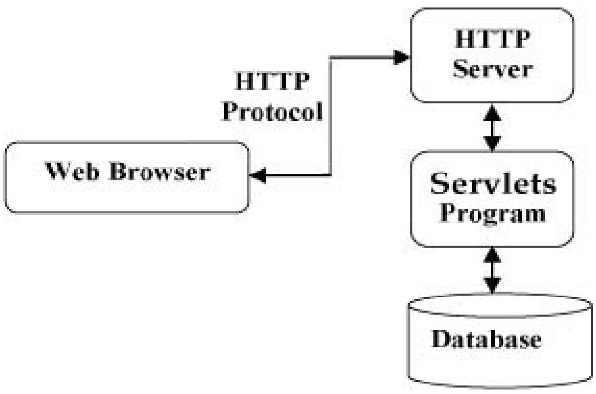
使用 Servlet，您可以收集来自网页表单的用户输入，呈现来自数据库或者其他源的记录，还可以动态创建网页。

Java Servlet 通常情况下与使用 CGI（Common Gateway Interface，公共网关接口）实现的程序可以达到异曲同工的效果。但是相比于 CGI，Servlet 有以下几点优势：

- 性能明显更好。
- Servlet 在 Web 服务器的地址空间内执行。这样它就没有必要再创建一个单独的进程来处理每个客户端请求。
- Servlet 是独立于平台的，因为它们是用 Java 编写的。
- 服务器上的 Java 安全管理器执行了一系列限制，以保护服务器计算机上的资源。因此，Servlet 是可信的。
- Java 类库的全部功能对 Servlet 来说都是可用的。它可以通过 sockets 和 RMI 机制与 applets、数据库或其他软件进行交互。

Servlet 架构

下图显示了 Servlet 在 Web 应用程序中的位置。



Servlet 任务

Servlet 执行以下主要任务：

- 读取客户端（浏览器）发送的显式的数据。这包括网页上的 HTML 表单，或者也可以是

来自 applet 或自定义的 HTTP 客户端程序的表单。
- 读取客户端（浏览器）发送的隐式的 HTTP 请求数据。这包括 cookies、媒体类型和浏览器能理解的压缩格式等等。
- 处理数据并生成结果。这个过程可能需要访问数据库，执行 RMI 或 CORBA 调用，调用 Web 服务，或者直接计算得出对应的响应。
- 发送显式的数据（即文档）到客户端（浏览器）。该文档的格式可以是多种多样的，包括文本文件（HTML 或 XML）、二进制文件（GIF 图像）、Excel 等。
- 发送隐式的 HTTP 响应到客户端（浏览器）。这包括告诉浏览器或其他客户端被返回的文档类型（例如 HTML），设置 cookies 和缓存参数，以及其他类似的任务。

Servlet 包

Java Servlet 是运行在带有支持 Java Servlet 规范的解释器的 web 服务器上的 Java 类。

Servlet 可以使用 **javax.servlet** 和 **javax.servlet.http** 包创建，它是 Java 企业版的标准组成部分，Java 企业版是支持大型开发项目的 Java 类库的扩展版本。

这些类实现 Java Servlet 和 JSP 规范。在写本教程的时候，二者相应的版本分别是 Java Servlet 2.5 和 JSP 2.1。

Java Servlet 就像任何其他的 Java 类一样已经被创建和编译。在您安装 Servlet 包并把它们添加到您的计算机上的 Classpath 类路径中之后，您就可以通过 JDK 的 Java 编译器或任何其他编译器来编译 Servlet。

下一步呢？

接下来，本教程会带你一步一步地设置您的 Servlet 环境，以便开始后续的 Servlet 使用。因此，请系紧您的安全带，随我们一起开始 Servlet 的学习之旅吧！相信您会很喜欢这个教程的。

Servlet 环境设置

开发环境是您可以开发、测试、运行 Servlet 的地方。

就像任何其他的 Java 程序，您需要通过使用 Java 编译器 **javac** 编译 Servlet，在编译 Servlet 应用程序后，将它部署在配置的环境中以便测试和运行。

这个开发环境设置包括以下步骤：

设置 Java 开发工具包（Java Development Kit）

这一步涉及到下载 Java 软件开发工具包（SDK，即 Software Development Kit），并适当地设置 PATH 环境变量。

您可以从 Oracle 的 Java 网站下载 SDK：Java SE Downloads。

一旦您下载了 SDK，请按照给定的指令来安装和配置设置。最后，设置 PATH 和 JAVA_HOME 环境变量指向包含 java 和 javac 的目录，通常分别为 java_install_dir/bin 和 java_install_dir。

如果您运行的是 Windows，并把 SDK 安装在 C:\jdk1.5.0_20 中，则需要在您的 C:\autoexec.bat 文件中放入下列的行：

```
set PATH=C:\jdk1.5.0_20\bin;%PATH%
set JAVA_HOME=C:\jdk1.5.0_20
```

或者，在 Windows NT/2000/XP 中，您也可以用鼠标右键单击"我的电脑"，选择"属性"，再选择"高级"，"环境变量"。然后，更新 PATH 的值，按下"确定"按钮。

在 Unix（Solaris、Linux 等）上，如果 SDK 安装在 /usr/local/jdk1.5.0_20 中，并且您使用的是 C shell，则需要在您的 .cshrc 文件中放入下列的行：

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

另外，如果您使用集成开发环境（IDE，即 Integrated Development Environment），比如 Borland JBuilder、Eclipse、IntelliJ IDEA 或 Sun ONE Studio，编译并运行一个简单的程序，以确认该 IDE 知道您安装的 Java 路径。

设置 Web 服务器：Tomcat

在市场上有许多 Web 服务器支持 Servlet。有些 Web 服务器是免费下载的，Tomcat 就是其中的一个。

Apache Tomcat 是一款 Java Servlet 和 JavaServer Pages 技术的开源软件实现，可以作为测试 Servlet 的独立服务器，而且可以集成到 Apache Web 服务器。下面是在电脑上安装 Tomcat 的步骤：

- 从 http://tomcat.apache.org/ 上下载最新版本的 Tomcat。
- 一旦您下载了 Tomcat，解压缩到一个方便的位置。例如，如果您使用的是 Windows，则解压缩到 C:\apache-tomcat-5.5.29 中，如果您使用的是 Linux/Unix，则解压缩到 /usr/local/apache-tomcat-5.5.29 中，并创建 CATALINA_HOME 环境变量指向这些位置。

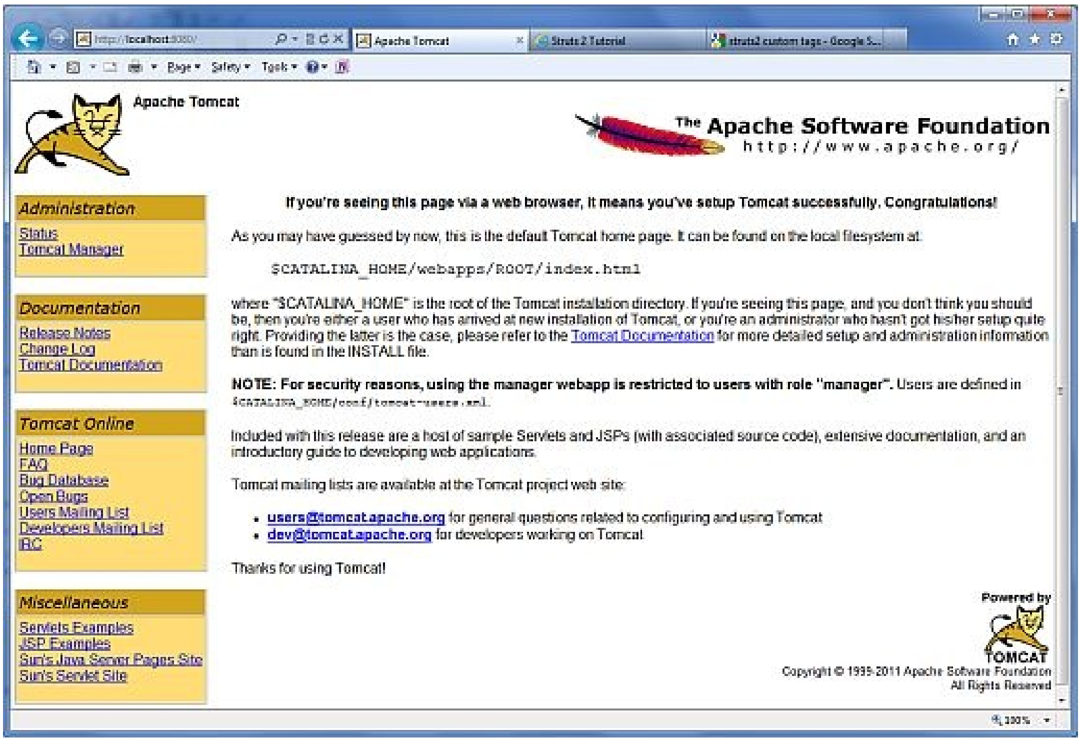
在 Windows 上，可以通过执行下面的命令来启动 Tomcat：

```
%CATALINA_HOME%\bin\startup.bat

or

C:\apache-tomcat-5.5.29\bin\startup.bat
```

在 Unix（Solaris、Linux 等） 上，可以通过执行下面的命令来启动 Tomcat：

```
$CATALINA_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-5.5.29/bin/startup.sh
```

Tomcat 启动后，可以通过在浏览器地址栏输入 **http://localhost:8080/** 访问 Tomcat 中的默认应用程序。如果一切顺利，那么会显示以下结果：



有关配置和运行 Tomcat 的进一步信息可以查阅应用程序安装的文档，或者可以访问 Tomcat 网站：http://tomcat.apache.org。

在 Windows 上，可以通过执行下面的命令来停止 Tomcat：

```
C:\apache-tomcat-5.5.29\bin\shutdown
```

在 Unix（Solaris、Linux 等） 上，可以通过执行下面的命令来停止 Tomcat：

```
/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

设置 CLASSPATH

由于 Servlet 不是 Java 平台标准版的组成部分，所以您必须为编译器指定 Servlet 类的路径。

如果您运行的是 Windows，则需要在您的 C:\autoexec.bat 文件中放入下列的行：

```
set CATALINA=C:\apache-tomcat-5.5.29
set CLASSPATH=%CATALINA%\common\lib\servlet-api.jar;%CLASSPATH%
```

或者，在 Windows NT/2000/XP 中，您也可以用鼠标右键单击"我的电脑"，选择"属性"，再选择"高级"，"环境变量"。然后，更新 CLASSPATH 的值，按下"确定"按钮。

在 Unix（Solaris、Linux 等）上，如果您使用的是 C shell，则需要在您的 .cshrc 文件中放入下列的行：

```
setenv CATALINA=/usr/local/apache-tomcat-5.5.29
setenv CLASSPATH $CATALINA/common/lib/servlet-api.jar:$CLASSPATH
```

注意：假设您的开发目录是 C:\ServletDevel（在 Windows 上）或 /user/ServletDevel（在 UNIX 上），那么您还需要在 CLASSPATH 中添加这些目录，添加方式与上面的添加方式类似。

Servlet 生命周期

Servlet 生命周期可被定义为从创建直到毁灭的整个过程。以下是 Servlet 遵循的过程：

- Servlet 通过调用 **init ()** 方法进行初始化。
- Servlet 调用 **service()** 方法来处理客户端的请求。
- Servlet 通过调用 **destroy()** 方法终止（结束）。
- 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

现在让我们详细讨论生命周期的方法。

init() 方法

init 方法被设计成只调用一次。它在第一次创建 Servlet 时被调用，在后续每次用户请求时不再调用。因此，它是用于一次性初始化，就像 Applet 的 init 方法一样。

Servlet 创建于用户第一次调用对应于该 Servlet 的 URL 时，但是您也可以指定 Servlet 在服务器第一次启动时被加载。

当用户调用一个 Servlet 时，就会创建一个 Servlet 实例，每一个用户请求都会产生一个新的线程，适当的时候移交给 doGet 或 doPost 方法。init() 方法简单地创建或加载一些数据，这些数据将被用于 Servlet 的整个生命周期。

init 方法的定义如下：

```
public void init() throws ServletException {
  // 初始化代码...
}
```

service() 方法

service() 方法是执行实际任务的主要方法。Servlet 容器（即 Web 服务器）调用 service() 方法来处理来自客户端（浏览器）的请求，并把格式化的响应写回给客户端。

每次服务器接收到一个 Servlet 请求时，服务器会产生一个新的线程并调用服务。service() 方法检查 HTTP 请求类型（GET、POST、PUT、DELETE 等），并在适当的时候调用 doGet、doPost、doPut，doDelete 等方法。

下面是该方法的特征：

```
public void service(ServletRequest request,
                    ServletResponse response)
      throws ServletException, IOException{
}
```

service() 方法由容器调用，service 方法在适当的时候调用 doGet、doPost、doPut、doDelete 等方法。所以，您不用对 service() 方法做任何动作，您只需要根据来自客户端的请求类型来重载 doGet() 或 doPost() 即可。

doGet() 和 doPost() 方法是每次服务请求中最常用的方法。下面是这两种方法的特征。

doGet() 方法

GET 请求来自于一个 URL 的正常请求，或者来自于一个未指定 METHOD 的 HTML 表单，它由 doGet() 方法处理。

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet 代码
}
```

doPost() 方法

POST 请求来自于一个特别指定了 METHOD 为 POST 的 HTML 表单，它由 doPost() 方法处理。

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet 代码
}
```

destroy() 方法

destroy() 方法只会被调用一次，在 Servlet 生命周期结束时被调用。destroy() 方法可以让您的 Servlet 关闭数据库连接、停止后台线程、把 Cookie 列表或点击计数器写入到磁盘，并执行其他类似的清理活动。

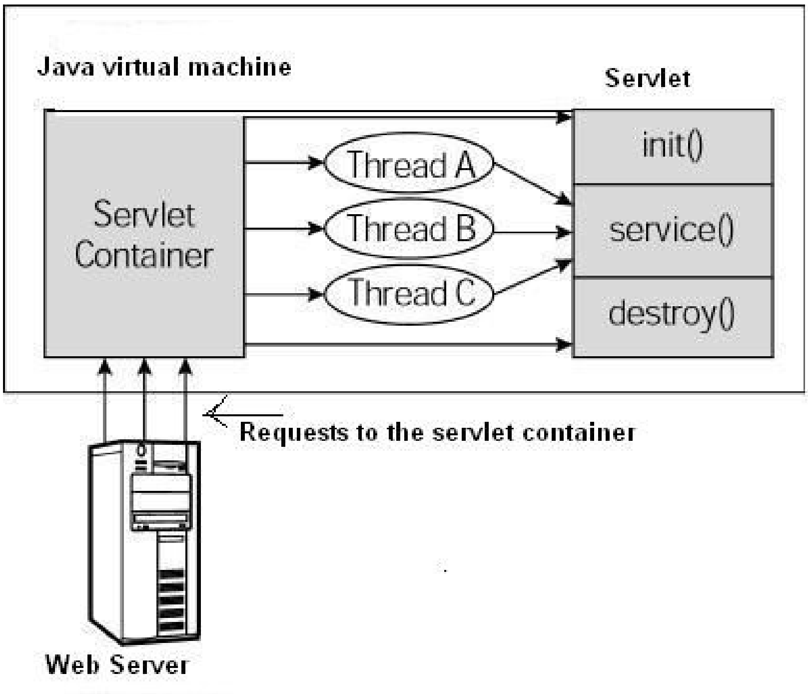
在调用 destroy() 方法之后，servlet 对象被标记为垃圾回收。destroy 方法定义如下所示：

```
  public void destroy() {
    // 终止化代码...
  }
```

架构图

下图显示了一个典型的 Servlet 生命周期方案。

- 第一个到达服务器的 HTTP 请求被委派到 Servlet 容器。
- Servlet 容器在调用 service() 方法之前加载 Servlet。
- 然后 Servlet 容器处理由多个线程产生的多个请求，每个线程执行一个单一的 Servlet 实例的 service() 方法。



Servlet 实例

Servlet 是服务 HTTP 请求并实现 **javax.servlet.Servlet** 接口的 Java 类。Web 应用程序开发人员通常编写 Servlet 来扩展 javax.servlet.http.HttpServlet，并实现 Servlet 接口的抽象类专门用来处理 HTTP 请求。

Hello World 示例代码

下面是 Servlet 输出 Hello World 的示例源代码：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloWorld extends HttpServlet {

  private String message;

  public void init() throws ServletException
  {
      // 执行必需的初始化
      message = "Hello World";
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      // 实际的逻辑是在这里
      PrintWriter out = response.getWriter();
      out.println("<h1>" + message + "</h1>");
  }

  public void destroy()
  {
      // 什么也不做
  }
}
```

编译 Servlet

让我们把上面的代码写在 HelloWorld.java 文件中，把这个文件放在 C:\ServletDevel（在 Windows 上）或 /usr/ServletDevel（在 UNIX 上）中，您还需要把这些目录添加到 CLASSPATH 中。

假设您的环境已经正确地设置，进入 **ServletDevel** 目录，并编译 HelloWorld.java，如下所示：

```
$ javac HelloWorld.java
```

如果 Servlet 依赖于任何其他库，您必须在 CLASSPATH 中包含那些 JAR 文件。在这里，我只包含了 servlet-api.jar JAR 文件，因为我没有在 Hello World 程序中使用任何其他库。

该命令行使用 Sun Microsystems Java 软件开发工具包（JDK）内置的 javac 编译器。为使该命令正常工作，您必须 PATH 环境变量中使用的 Java SDK 的位置。

如果一切顺利，上面编译会在同一目录下生成 HelloWorld.class 文件。下一节将讲解已编译的 Servlet 如何部署在生产中。

Servlet 部署

默认情况下，Servlet 应用程序位于路径 <Tomcat-installation-directory>/webapps/ROOT 下，且类文件放在 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

如果您有一个完全合格的类名称 **com.myorg.MyServlet**，那么这个 Servlet 类必须位于 WEB-INF/classes/com/myorg/MyServlet.class 中。

现在，让我们把 HelloWorld.class 复制到 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中，并在位于 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/ 的 **web.xml** 文件中创建以下条目：

```
    <servlet>
        <servlet-name>HelloWorld</servlet-name>
        <servlet-class>HelloWorld</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/HelloWorld</url-pattern>
    </servlet-mapping>
```

上面的条目要被创建在 web.xml 文件中的 <web-app>...</web-app> 标签内。在该文件中可能已经有各种可用的条目，但不要在意。

到这里，您基本上已经完成了，现在让我们使用 <Tomcat-installation-directory>\bin\startup.bat（在 Windows 上）或 <Tomcat-installation-directory>/bin/startup.sh（在 Linux/Solaris 等上）启动 tomcat 服务器，最后在浏览器的地址栏中输入 **http://localhost:8080/HelloWorld**。如果一切顺利，您会看到下面的结果：

http://localhost:8080/HelloWorld - Windows Internet Explorer
http://localhost:8080/HelloWorld
File Edit View Favorites Tools Help
http://localhost:8080/HelloWorld
Hello World

Servlet 表单数据

很多情况下，需要传递一些信息，从浏览器到 Web 服务器，最终到后台程序。浏览器使用两种方法可将这些信息传递到 Web 服务器，分别为 GET 方法和 POST 方法。

GET 方法

GET 方法向页面请求发送已编码的用户信息。页面和已编码的信息中间用 ? 字符分隔，如下所示：

```
http://www.test.com/hello?key1=value1&key2=value2
```

GET 方法是默认的从浏览器向 Web 服务器传递信息的方法，它会产生一个很长的字符串，出现在浏览器的地址栏中。如果您要向服务器传递的是密码或其他的敏感信息，请不要使用 GET 方法。GET 方法有大小限制：请求字符串中最多只能有 1024 个字符。

这些信息使用 QUERY_STRING 头传递，并可以通过 QUERY_STRING 环境变量访问，Servlet 使用 **doGet()** 方法处理这种类型的请求。

POST 方法

另一个向后台程序传递信息的比较可靠的方法是 POST 方法。POST 方法打包信息的方式与 GET 方法基本相同，但是 POST 方法不是把信息作为 URL 中 ? 字符后的文本字符串进行发送，而是把这些信息作为一个单独的消息。消息以标准输出的形式传到后台程序，您可以解析和使用这些标准输出。Servlet 使用 doPost() 方法处理这种类型的请求。

使用 Servlet 读取表单数据

Servlet 处理表单数据，这些数据会根据不同的情况使用不同的方法自动解析：

- **getParameter()**：您可以调用 request.getParameter() 方法来获取表单参数的值。
- **getParameterValues()**：如果参数出现一次以上，则调用该方法，并返回多个值，例如复选框。
- **getParameterNames()**：如果您想要得到当前请求中的所有参数的完整列表，则调用该方法。

使用 URL 的 GET 方法实例

下面是一个简单的 URL，将使用 GET 方法向 HelloForm 程序传递两个值。

http://localhost:8080/HelloForm?first_name=ZARA&last_name=ALI

下面是处理 Web 浏览器输入的 **HelloForm.java** Servlet 程序。我们将使用 **getParameter()** 方法，可以很容易地访问传递的信息：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloForm extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "使用 GET 方法读取表单数据";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>名字</b>："
                + request.getParameter("first_name") + "\n" +
                "  <li><b>姓氏</b>："
                + request.getParameter("last_name") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
}
```

假设您的环境已经正确地设置，编译 HelloForm.java，如下所示：

```
$ javac HelloForm.java
```

如果一切顺利，上述编译会产生 **HelloForm.class** 文件。接下来，您就必须把该类文件复制到 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中，并在位于 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/ 的 **web.xml** 文件中创建以下条目：

```
    <servlet>
        <servlet-name>HelloForm</servlet-name>
        <servlet-class>HelloForm</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloForm</servlet-name>
        <url-pattern>/HelloForm</url-pattern>
    </servlet-mapping>
```

现在在浏览器的地址栏中输入 *http://localhost:8080/HelloForm?first_name=ZARA&last_name=ALI*，并在触发上述命令之前确保已经启动 Tomcat 服务器。如果一切顺利，您会得到下面的结果：

```
<h1>使用 GET 方法读取表单数据</h1>

<ul>
<li><b>名字<b>：ZARA</li>
<li><b>姓氏<b>：ALI</li>
</ul>
```

使用表单的 GET 方法实例

下面是一个简单的实例，使用 HTML 表单和提交按钮传递两个值。我们将使用相同的 Servlet HelloForm 来处理输入。

```
<html>
<body>
<form action="HelloForm" method="GET">
名字：<input type="text" name="first_name">
<br />
姓氏：<input type="text" name="last_name" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

保存这个 HTML 到 hello.htm 文件中，并把它放在 <Tomcat-installation-directory>/webapps/ROOT 目录下。当您访问 *http://localhost:8080/Hello.htm* 时，下面是上面表单的实际输出。

```
<form action="javascript:void();" method="get" target="_blank">名字：<input type="text" na
姓氏：<input type="text" name="last_name"> <input type="button" value="提交"></form>
```

尝试输入名字和姓氏，然后点击"提交"按钮，在您本机上查看输出结果。基于所提供的输入，它会产生与上一个实例类似的结果。

使用表单的 POST 方法实例

让我们对上面的 Servlet 做小小的修改，以便它可以处理 GET 和 POST 方法。下面的 **HelloForm.java** Servlet 程序使用 GET 和 POST 方法处理由 Web 浏览器给出的输入。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloForm extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Using GET Method to Read Form Data";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>名字</b>："
                + request.getParameter("first_name") + "\n" +
                "  <li><b>姓氏</b>："
                + request.getParameter("last_name") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，编译部署上述的 Servlet，并使用带有 POST 方法的 Hello.htm 进行测试，如下所示：

```
<html>
<body>
<form action="HelloForm" method="POST">
名字：<input type="text" name="first_name">
<br />
姓氏：<input type="text" name="last_name" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

下面是上面表单的实际输出，尝试输入名字和姓氏，然后点击"提交"按钮，在您本机上查看输出结果。

```
<form action="javascript:void();" method="get" target="_blank">名字：<input type="text" na
姓氏：<input type="text" name="last_name"> <input type="button" value="提交"></form>
```

基于所提供的输入，它会产生与上一个实例类似的结果。

将复选框数据传递到 Servlet 程序

当需要选择一个以上的选项时，则使用复选框。

下面是一个 HTML 代码实例 CheckBox.htm，一个带有两个复选框的表单。

```
<html>
<body>
<form action="CheckBox" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> 数学
<input type="checkbox" name="physics"  /> 物理
<input type="checkbox" name="chemistry" checked="checked" />
                                                     化学
<input type="submit" value="选择学科" />
</form>
</body>
</html>
```

这段代码的结果是下面的表单：

```
<form action="javascript:void();" method="get" target="_blank"><input type="checkbox" nam
```

下面是 CheckBox.java Servlet 程序，处理 Web 浏览器给出的复选框输入。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class CheckBox extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "读取复选框数据";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>数学标识：</b>: "
                + request.getParameter("maths") + "\n" +
                "  <li><b>物理标识：</b>: "
                + request.getParameter("physics") + "\n" +
                "  <li><b>化学标识：</b>: "
                + request.getParameter("chemistry") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

上面的实例将显示下面的结果：

```
<h1>读取复选框数据</h1>

<ul>
<li><b>数学标识：</b>on</li>
<li><b>物理标识：</b>null</li>
<li><b>化学标识：</b>on</li>
</ul>
```

读取所有的表单参数

以下是通用的实例，使用 HttpServletRequest 的 **getParameterNames()** 方法读取所有可用的表单参数。该方法返回一个枚举，其中包含未指定顺序的参数名。

一旦我们有一个枚举，我们可以以标准方式循环枚举，使用 *hasMoreElements()* 方法来确定何时停止，使用 *nextElement()* 方法来获取每个参数的名称。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class ReadParams extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "读取所有的表单数据";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<table width=\"100%\" border=\"1\" align=\"center\">\n" +
        "<tr bgcolor=\"#949494\">\n" +
        "<th>参数名称</th><th>参数值</th>\n"+
        "</tr>\n");

      Enumeration paramNames = request.getParameterNames();

      while(paramNames.hasMoreElements()) {
         String paramName = (String)paramNames.nextElement();
         out.print("<tr><td>" + paramName + "</td>\n<td>");
         String[] paramValues =
                request.getParameterValues(paramName);
         // 读取单个值的数据
         if (paramValues.length == 1) {
           String paramValue = paramValues[0];
           if (paramValue.length() == 0)
             out.println("<i>No Value</i>");
           else
             out.println(paramValue);
         } else {
             // 读取多个值的数据
             out.println("<ul>");
             for(int i=0; i < paramValues.length; i++) {
                out.println("<li>" + paramValues[i]);
             }
             out.println("</ul>");
         }
      }
      out.println("</tr>\n</table>\n</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，通过下面的表单尝试上面的 Servlet：

```
<html>
<body>
<form action="ReadParams" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> 数学
<input type="checkbox" name="physics"  /> 物理
<input type="checkbox" name="chemistry" checked="checked" /> 化学
<input type="submit" value="选择学科" />
</form>
</body>
</html>
```

现在使用上面的表单调用 Servlet，将产生以下结果：

读取所有的表单数据

参数名称	参数值
maths	on
chemistry	on

您可以尝试使用上面的 Servlet 来读取其他的表单数据，比如文本框、单选按钮或下拉框等。

Servlet 客户端 HTTP 请求

当浏览器请求网页时，它会向 Web 服务器发送特定信息，这些信息不能被直接读取，因为这些信息是作为 HTTP 请求的头的一部分进行传输的。您可以查看 HTTP 协议 了解更多相关信息。

以下是来自于浏览器端的重要头信息，您可以在 Web 编程中频繁使用：

头信息	描述
Accept	这个头信息指定浏览器或其他客户端可以处理的 MIME 类型。值 **image/png** 或 **image/jpeg** 是最常见的两种可能值。
Accept-Charset	这个头信息指定浏览器可以用来显示信息的字符集。例如 ISO-8859-1。
Accept-Encoding	这个头信息指定浏览器知道如何处理的编码类型。值 **gzip** 或 **compress** 是最常见的两种可能值。
Accept-Language	这个头信息指定客户端的首选语言，在这种情况下，Servlet 会产生多种语言的结果。例如，en、en-us、ru 等。
Authorization	这个头信息用于客户端在访问受密码保护的网页时识别自己的身份。
Connection	这个头信息指示客户端是否可以处理持久 HTTP 连接。持久连接允许客户端或其他浏览器通过单个请求来检索多个文件。值 **Keep-Alive** 意味着使用了持续连接。
Content-Length	这个头信息只适用于 POST 请求，并给出 POST 数据的大小（以字节为单位）。
Cookie	这个头信息把之前发送到浏览器的 cookies 返回到服务器。
Host	这个头信息指定原始的 URL 中的主机和端口。
If-Modified-Since	这个头信息表示只有当页面在指定的日期后已更改时，客户端想要的页面。如果没有新的结果可以使用，服务器会发送一个 304 代码，表示 **Not Modified** 头信息。
If-Unmodified-Since	这个头信息是 If-Modified-Since 的对立面，它指定只有当文档早于指定日期时，操作才会成功。
Referer	这个头信息指示所指向的 Web 页的 URL。例如，如果您在网页 1，点击一个链接到网页 2，当浏览器请求网页 2 时，网页 1 的 URL 就会包含在 Referer 头信息中。
User-Agent	这个头信息识别发出请求的浏览器或其他客户端，并可以向不同类型的浏览器返回不同的内容。

读取 HTTP 头的方法

下面的方法可用在 Servlet 程序中读取 HTTP 头。这些方法通过 *HttpServletRequest* 对象可用。

方法	描述
Cookie[] getCookies()	返回一个数组，包含客户端发送该请求的所有的 Cookie 对象。
Enumeration getAttributeNames()	返回一个枚举，包含提供给该请求可用的属性名称。
Enumeration getHeaderNames()	返回一个枚举，包含在该请求中包含的所有的头名。
Enumeration getParameterNames()	返回一个 String 对象的枚举，包含在该请求中包含的参数的名称。
HttpSession getSession()	返回与该请求关联的当前 session 会话，或者如果请求没有 session 会话，则创建一个。
HttpSession getSession(boolean create)	返回与该请求关联的当前 HttpSession，或者如果没有当前会话，且创建是真的，则返回一个新的 session 会话。
Locale getLocale()	基于 Accept-Language 头，返回客户端接受内容的首选的区域设置。
Object getAttribute(String name)	以对象形式返回已命名属性的值，如果没有给定名称的属性存在，则返回 null。
ServletInputStream getInputStream()	使用 ServletInputStream，以二进制数据形式检索请求的主体。
String getAuthType()	返回用于保护 Servlet 的身份验证方案的名称，例如，"BASIC" 或 "SSL"，如果JSP没有受到保护则返回 null。
String getCharacterEncoding()	返回请求主体中使用的字符编码的名称。
String getContentType()	返回请求主体的 MIME 类型，如果不知道类型则返回 null。
String getContextPath()	返回指示请求上下文的请求 URI 部分。
String getHeader(String name)	以字符串形式返回指定的请求头的值。
String getMethod()	返回请求的 HTTP 方法的名称，例如，GET、POST 或 PUT。
String getParameter(String name)	以字符串形式返回请求参数的值，或者如果参数不存在则返回 null。
String getPathInfo()	当请求发出时，返回与客户端发送的 URL 相关的任何额外的路径信息。
String getProtocol()	返回请求协议的名称和版本。

String getQueryString()	返回包含在路径后的请求 URL 中的查询字符串。
String getRemoteAddr()	返回发送请求的客户端的互联网协议（IP）地址。
String getRemoteHost()	返回发送请求的客户端的完全限定名称。
String getRemoteUser()	如果用户已通过身份验证，则返回发出请求的登录用户，或者如果用户未通过身份验证，则返回 null。
String getRequestURI()	从协议名称直到 HTTP 请求的第一行的查询字符串中，返回该请求的 URL 的一部分。
String getRequestedSessionId()	返回由客户端指定的 session 会话 ID。
String getServletPath()	返回调用 JSP 的请求的 URL 的一部分。
String[] getParameterValues(String name)	返回一个字符串对象的数组，包含所有给定的请求参数的值，如果参数不存在则返回 null。
boolean isSecure()	返回一个布尔值，指示请求是否使用安全通道，如 HTTPS。
int getContentLength()	以字节为单位返回请求主体的长度，并提供输入流，或者如果长度未知则返回 -1。
int getIntHeader(String name)	返回指定的请求头的值为一个 int 值。
int getServerPort()	返回接收到这个请求的端口号。

HTTP Header 请求实例

下面的实例使用 HttpServletRequest 的 **getHeaderNames()** 方法读取 HTTP 头信息。该方法返回一个枚举，包含与当前的 HTTP 请求相关的头信息。

一旦我们有一个枚举，我们可以以标准方式循环枚举，使用 *hasMoreElements()* 方法来确定何时停止，使用 *nextElement()* 方法来获取每个参数的名称。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class DisplayHeader extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "HTTP Header 请求实例";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n"+
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<table width=\"100%\" border=\"1\" align=\"center\">\n" +
        "<tr bgcolor=\"#949494\">\n" +
        "<th>Header 名称</th><th>Header 值</th>\n"+
        "</tr>\n");

      Enumeration headerNames = request.getHeaderNames();

      while(headerNames.hasMoreElements()) {
         String paramName = (String)headerNames.nextElement();
         out.print("<tr><td>" + paramName + "</td>\n");
         String paramValue = request.getHeader(paramName);
         out.println("<td> " + paramValue + "</td></tr>\n");
      }
      out.println("</table>\n</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，调用上面的 Servlet 会产生以下结果：

```
<h1>HTTP Header 请求实例</h1>

<table>

<tbody>

<tr bgcolor="#949494"><th>Header 名称</th><th>Header 值</th></tr>

<tr><td>accept</td><td>*/*</td></tr>

<tr><td>accept-language</td><td>en-us</td></tr>

<tr><td>user-agent</td><td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0

<tr><td>accept-encoding</td><td>gzip, deflate</td></tr>

<tr><td>host</td><td>localhost:8080</td></tr>

<tr><td>connection</td><td>Keep-Alive</td></tr>

<tr><td>cache-control</td><td>no-cache</td></tr>

</tbody>

</table>
```

Servlet 服务器 HTTP 响应

正如前面的章节中讨论的那样，当一个 Web 服务器响应一个 HTTP 请求时，响应通常包括一个状态行、一些响应报头、一个空行和文档。一个典型的响应如下所示：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包括 HTTP 版本（在本例中为 HTTP/1.1）、一个状态码（在本例中为 200）和一个对应于状态码的短消息（在本例中为 OK）。

下表总结了从 Web 服务器端返回到浏览器的最有用的 HTTP 1.1 响应报头，您会在 Web 编程中频繁地使用它们：

头信息	描述
Allow	这个头信息指定服务器支持的请求方法（GET、POST 等）。
Cache-Control	这个头信息指定响应文档在何种情况下可以安全地缓存。可能的值有：**public**、**private** 或 **no-cache** 等。Public 意味着文档是可缓存，Private 意味着文档是单个用户私用文档，且只能存储在私有（非共享）缓存中，no-cache 意味着文档不应被缓存。
Connection	这个头信息指示浏览器是否使用持久 HTTP 连接。值 **close** 指示浏览器不使用持久 HTTP 连接，值 **keep-alive** 意味着使用持久连接。
Content-Disposition	这个头信息可以让您请求浏览器要求用户以给定名称的文件把响应保存到磁盘。
Content-Encoding	在传输过程中，这个头信息指定页面的编码方式。
Content-Language	这个头信息表示文档编写所使用的语言。例如，en、en-us、ru 等。
Content-Length	这个头信息指示响应中的字节数。只有当浏览器使用持久（keep-alive）HTTP 连接时才需要这些信息。
Content-Type	这个头信息提供了响应文档的 MIME（Multipurpose Internet Mail Extension）类型。
Expires	这个头信息指定内容过期的时间，在这之后内容不再被缓存。
Last-Modified	这个头信息指示文档的最后修改时间。然后，客户端可以缓存文件，并在以后的请求中通过 **If-Modified-Since** 请求头信息提供一个日期。
Location	这个头信息应被包含在所有的带有状态码的响应中。在 300s 内，这会通知浏览器文档的地址。浏览器会自动重新连接到这个位置，并获取新的文档。
Refresh	这个头信息指定浏览器应该如何尽快请求更新的页面。您可以指定页面刷新的秒数。
Retry-After	这个头信息可以与 503（Service Unavailable 服务不可用）响应配合使用，这会告诉客户端多久就可以重复它的请求。
Set-Cookie	个头信息指定一个与页面关联的 cookie。

设置 HTTP 响应报头的方法

下面的方法可用于在 Servlet 程序中设置 HTTP 响应报头。这些方法通过 *HttpServletResponse* 对象可用。

方法	描述
String encodeRedirectURL(String url)	为 sendRedirect 方法中使用的指定的 URL 进行编码，或者如果编码不是必需的，则返回 URL 未改变。

String encodeURL(String url)	对包含 session 会话 ID 的指定 URL 进行编码，或者如果编码不是必需的，则返回 URL 未改变。
boolean containsHeader(String name)	返回一个布尔值，指示是否已经设置已命名的响应报头。
boolean isCommitted()	返回一个布尔值，指示响应是否已经提交。
void addCookie(Cookie cookie)	把指定的 cookie 添加到响应。
void addDateHeader(String name, long date)	添加一个带有给定的名称和日期值的响应报头。
void addHeader(String name, String value)	添加一个带有给定的名称和值的响应报头。
void addIntHeader(String name, int value)	添加一个带有给定的名称和整数值的响应报头。
void flushBuffer()	强制任何在缓冲区中的内容被写入到客户端。
void reset()	清除缓冲区中存在的任何数据，包括状态码和头。
void resetBuffer()	清除响应中基础缓冲区的内容，不清除状态码和头。
void sendError(int sc)	使用指定的状态码发送错误响应到客户端，并清除缓冲区。
void sendError(int sc, String msg)	使用指定的状态发送错误响应到客户端。
void sendRedirect(String location)	使用指定的重定向位置 URL 发送临时重定向响应到客户端。
void setBufferSize(int size)	为响应主体设置首选的缓冲区大小。
void setCharacterEncoding(String charset)	设置被发送到客户端的响应的字符编码（MIME 字符集）例如，UTF-8。
void setContentLength(int len)	设置在 HTTP Servlet 响应中的内容主体的长度，该方法设置 HTTP Content-Length 头。
void setContentType(String type)	如果响应还未被提交，设置被发送到客户端的响应的内容类型。
void setDateHeader(String name, long date)	设置一个带有给定的名称和日期值的响应报头。
void setHeader(String name, String value)	设置一个带有给定的名称和值的响应报头。
void setIntHeader(String name, int value)	设置一个带有给定的名称和整数值的响应报头。
void setLocale(Locale loc)	如果响应还未被提交，设置响应的区域。
void setStatus(int sc)	为该响应设置状态码。

HTTP Header 响应实例

您已经在前面的实例中看到 setContentType() 方法，下面的实例也使用了同样的方法，此外，我们会用 **setIntHeader()** 方法来设置 **Refresh** 头。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class Refresh extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置刷新自动加载时间为 5 秒
      response.setIntHeader("Refresh", 5);

      // 设置响应内容类型
      response.setContentType("text/html");

      // Get current time
      Calendar calendar = new GregorianCalendar();
      String am_pm;
      int hour = calendar.get(Calendar.HOUR);
      int minute = calendar.get(Calendar.MINUTE);
      int second = calendar.get(Calendar.SECOND);
      if(calendar.get(Calendar.AM_PM) == 0)
        am_pm = "AM";
      else
        am_pm = "PM";

      String CT = hour+":"+ minute +":"+ second +" "+ am_pm;

      PrintWriter out = response.getWriter();
      String title = "自动刷新 Header 设置";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n"+
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<p>当前时间是：" + CT + "</p>\n");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，调用上面的 Servlet，每隔 5 秒会显示当前系统时间。只要运行 Servlet 并稍等片刻，即可看到如下的结果：

```
<h1>自动刷新 Header 设置</h1>

当前时间是：9:44:50 PM
```

Servlet HTTP 状态码

HTTP 请求和 HTTP 响应消息的格式是类似的，结构如下：

- 初始状态行 + 回车换行符（回车+换行）
- 零个或多个标题行+回车换行符
- 一个空白行，即回车换行符
- 一个可选的消息主体，比如文件、查询数据或查询输出

例如，服务器的响应头如下所示：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包括 HTTP 版本（在本例中为 HTTP/1.1）、一个状态码（在本例中为 200）和一个对应于状态码的短消息（在本例中为 OK）。

以下是可能从 Web 服务器返回的 HTTP 状态码和相关的信息列表：

代码	消息	描述
100	Continue	只有请求的一部分已经被服务器接收，但只要它没有被拒绝，客户端应继续该请求。
101	Switching Protocols	服务器切换协议。
200	OK	请求成功。
201	Created	该请求是完整的，并创建一个新的资源。
202	Accepted	该请求被接受处理，但是该处理是不完整的。
203	Non-authoritative Information	
204	No Content	
205	Reset Content	

206	Partial Content	
300	Multiple Choices	链接列表。用户可以选择一个链接，进入到该位置。最多五个地址。
301	Moved Permanently	所请求的页面已经转移到一个新的 URL。
302	Found	所请求的页面已经临时转移到一个新的 URL。
303	See Other	所请求的页面可以在另一个不同的 URL 下被找到。
304	Not Modified	
305	Use Proxy	
306	*Unused*	在以前的版本中使用该代码。现在已不再使用它，但代码仍被保留。
307	Temporary Redirect	所请求的页面已经临时转移到一个新的 URL。
400	Bad Request	服务器不理解请求。
401	Unauthorized	所请求的页面需要用户名和密码。
402	Payment Required	您还不能使用该代码。
403	Forbidden	禁止访问所请求的页面。
404	Not Found	服务器无法找到所请求的页面。.
405	Method Not Allowed	在请求中指定的方法是不允许的。
406	Not Acceptable	服务器只生成一个不被客户端接受的响应。
407	Proxy Authentication Required	在请求送达之前，您必须使用代理服务器的验证。
408	Request Timeout	请求需要的时间比服务器能够等待的时间长，超时。
409	Conflict	请求因为冲突无法完成。
410	Gone	所请求的页面不再可用。
411	Length Required	"Content-Length" 未定义。服务器无法处理客户端发送的不带 Content-Length 的请求信息。
412	Precondition Failed	请求中给出的先决条件被服务器评估为 false。
413	Request Entity Too Large	服务器不接受该请求，因为请求实体过大。
414	Request-url Too Long	服务器不接受该请求，因为 URL 太长。当您转换一个 "post" 请求为一个带有长的查询信息的 "get" 请求时发生。

415	Unsupported Media Type	服务器不接受该请求，因为媒体类型不被支持。
417	Expectation Failed	
500	Internal Server Error	未完成的请求。服务器遇到了一个意外的情况。
501	Not Implemented	未完成的请求。服务器不支持所需的功能。
502	Bad Gateway	未完成的请求。服务器从上游服务器收到无效响应。
503	Service Unavailable	未完成的请求。服务器暂时超载或死机。
504	Gateway Timeout	网关超时。
505	HTTP Version Not Supported	服务器不支持"HTTP协议"版本。

设置 HTTP 状态代码的方法

下面的方法可用于在 Servlet 程序中设置 HTTP 状态码。这些方法通过 *HttpServletResponse* 对象可用。

方法	描述
public void setStatus (int statusCode)	该方法设置一个任意的状态码。setStatus 方法接受一个 int（状态码）作为参数。如果您的反应包含了一个特殊的状态码和文档，请确保在使用 *PrintWriter* 实际返回任何内容之前调用 setStatus。
public void sendRedirect(String url)	该方法生成一个 302 响应，连同一个带有新文档 URL 的 *Location* 头。
public void sendError(int code, String message)	该方法发送一个状态码（通常为 404），连同一个在 HTML 文档内部自动格式化并发送到客户端的短消息。

HTTP 状态码实例

下面的例子把 407 错误代码发送到客户端浏览器，浏览器会显示 "Need authentication!!!" 消息。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class showError extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置错误代码和原因
      response.sendError(407, "Need authentication!!!" );
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，调用上面的 Servlet 将显示以下结果：

```
<h2>HTTP Status 407 - Need authentication!!!</h2>
<b>type</b> Status report
<b>message</b> <u>Need authentication!!!</u>
<b>description</b> <u>The client must first authenticate itself with the proxy (Need auth
<h3>Apache Tomcat/5.5.29</h3>
```

Servlet 编写过滤器

Servlet 过滤器是可用于 Servlet 编程的 Java 类，有以下目的：

- 在客户端的请求访问后端资源之前，拦截这些请求。
- 在服务器的响应发送回客户端之前，处理这些响应。

根据规范建议的各种类型的过滤器：

- 身份验证过滤器（Authentication Filters）。
- 数据压缩过滤器（Data compression Filters）。
- 加密过滤器（Encryption Filters）。
- 触发资源访问事件过滤器。
- 图像转换过滤器（Image Conversion Filters）。
- 日志记录和审核过滤器（Logging and Auditing Filters）。
- MIME-TYPE 链过滤器（MIME-TYPE Chain Filters）。
- 标记化过滤器（Tokenizing Filters）。
- XSL/T 过滤器（XSL/T Filters），转换 XML 内容。

过滤器被部署在部署描述符文件 **web.xml** 中，然后映射到您的应用程序的部署描述符中的 Servlet 名称或 URL 模式。

当 Web 容器启动 Web 应用程序时，它会为您在部署描述符中声明的每一个过滤器创建一个实例。该过滤器执行的顺序是按它们在部署描述符中声明的顺序。

Servlet 过滤器方法

过滤器是一个实现了 javax.servlet.Filter 接口的 Java 类。javax.servlet.Filter 接口定义了三个方法：

方法	描述
public void doFilter (ServletRequest, ServletResponse, FilterChain)	该方法在每次一个请求/响应对因客户端在链的末端请求资源而通过链传递时由容器调用。
public void init(FilterConfig filterConfig)	该方法由 Web 容器调用，指示一个过滤器被放入服务。
public void destroy()	该方法由 Web 容器调用，指示一个过滤器被取出服务。

Servlet 过滤器实例

以下是 Servlet 过滤器的实例，将输出客户端的 IP 地址和当前的日期时间。本实例让您对 Servlet 过滤器有基本的了解，您可以使用相同的概念编写更复杂的过滤器应用程序：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 实现 Filter 类
public class LogFilter implements Filter  {
   public void  init(FilterConfig config)
                         throws ServletException{
      // 获取初始化参数
      String testParam = config.getInitParameter("test-param");

      // 输出初始化参数
      System.out.println("Test Param: " + testParam);
   }
   public void  doFilter(ServletRequest request,
                 ServletResponse response,
                 FilterChain chain)
                 throws java.io.IOException, ServletException {

      // 获取客户机的 IP 地址
      String ipAddress = request.getRemoteAddr();

      // 记录 IP 地址和当前时间戳
      System.out.println("IP "+ ipAddress + ", Time "
                                     + new Date().toString());

      // 把请求传回过滤链
      chain.doFilter(request,response);
   }
   public void destroy( ){
      /* 在 Filter 实例被 Web 容器从服务移除之前调用 */
   }
}
```

以通常的方式编译 **LogFilter.java**，把您的类文件放入 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

Web.xml 中的 Servlet 过滤器映射（Servlet Filter Mapping）

定义过滤器，然后映射到一个 URL 或 Servlet，这与定义 Servlet，然后映射到一个 URL 模式方式大致相同。在部署描述符文件 **web.xml** 中为 filter 标签创建下面的条目：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

上述过滤器适用于所有的 Servlet，因为我们在配置中指定 */** 。如果您只想在少数的 Servlet 上应用过滤器，您可以指定一个特定的 Servlet 路径。

现在试着以常用的方式调用任何 Servlet，您将会看到在 Web 服务器中生成的日志。您也可以使用 Log4J 记录器来把上面的日志记录到一个单独的文件中。

使用多个过滤器

Web 应用程序可以根据特定的目的定义若干个不同的过滤器。假设您定义了两个过滤器 *AuthenFilter* 和 *LogFilter*。您需要创建一个如下所述的不同的映射，其余的处理与上述所讲解的大致相同：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter>
   <filter-name>AuthenFilter</filter-name>
   <filter-class>AuthenFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

过滤器的应用顺序

web.xml 中的 filter-mapping 元素的顺序决定了 Web 容器应用过滤器到 Servlet 的顺序。若要反转过滤器的顺序，您只需要在 web.xml 文件中反转 filter-mapping 元素即可。

例如，上面的实例将先应用 LogFilter，然后再应用 AuthenFilter，但是下面的实例将颠倒这个顺序：

```
<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Servlet 异常处理

当一个 Servlet 抛出一个异常时，Web 容器在使用了 exception-type 元素的 **web.xml** 中搜索与抛出异常类型相匹配的配置。

您必须在 web.xml 中使用 **error-page** 元素来指定对特定异常 或 HTTP 状态码 作出相应的 Servlet 调用。

web.xml 配置

假设，有一个 *ErrorHandler* 的 Servelt 在任何已定义的异常或错误出现时被调用。以下将是在 web.xml 中创建的项。

```
<!-- servlet 定义 -->
<servlet>
        <servlet-name>ErrorHandler</servlet-name>
        <servlet-class>ErrorHandler</servlet-class>
</servlet>
<!-- servlet 映射 -->
<servlet-mapping>
        <servlet-name>ErrorHandler</servlet-name>
        <url-pattern>/ErrorHandler</url-pattern>
</servlet-mapping>

<!-- error-code 相关的错误页面 -->
<error-page>
    <error-code>404</error-code>
    <location>/ErrorHandler</location>
</error-page>
<error-page>
    <error-code>403</error-code>
    <location>/ErrorHandler</location>
</error-page>

<!-- exception-type 相关的错误页面 -->
<error-page>
    <exception-type>
          javax.servlet.ServletException
    </exception-type >
    <location>/ErrorHandler</location>
</error-page>

<error-page>
    <exception-type>java.io.IOException</exception-type >
    <location>/ErrorHandler</location>
</error-page>
```

如果您想对所有的异常有一个通用的错误处理程序，那么应该定义下面的 error-page，而不是为每个异常定义单独的 error-page 元素：

```
<error-page>
    <exception-type>java.lang.Throwable</exception-type >
    <location>/ErrorHandler</location>
</error-page>
```

以下是关于上面的 web.xml 异常处理要注意的点：

- Servelt ErrorHandler 与其他的 Servelt 的定义方式一样，且在 web.xml 中进行配置。
- 如果有错误状态代码出现，不管为 404（Not Found 未找到）或 403（Forbidden 禁止），则会调用 ErrorHandler 的 Servlet。
- 如果 Web 应用程序抛出 *ServletException* 或 *IOException*，那么 Web 容器会调用 ErrorHandler 的 Servlet。
- 您可以定义不同的错误处理程序来处理不同类型的错误或异常。上面的实例是非常通用的，希望您能通过实例理解基本的概念。

请求属性 - 错误/异常

以下是错误处理的 Servlet 可以访问的请求属性列表，用来分析错误/异常的性质。

属性	描述
javax.servlet.error.status_code	该属性给出状态码，状态码可被存储，并在存储为 java.lang.Integer 数据类型后可被分析。
javax.servlet.error.exception_type	该属性给出异常类型的信息，异常类型可被存储，并在存储为 java.lang.Class 数据类型后可被分析。
javax.servlet.error.message	该属性给出确切错误消息的信息，信息可被存储，并在存储为 java.lang.String 数据类型后可被分析。
javax.servlet.error.request_uri	该属性给出有关 URL 调用 Servlet 的信息，信息可被存储，并在存储为 java.lang.String 数据类型后可被分析。
javax.servlet.error.exception	该属性给出异常产生的信息，信息可被存储，并在存储为 java.lang.Throwable 数据类型后可被分析。
javax.servlet.error.servlet_name	该属性给出 Servlet 的名称，名称可被存储，并在存储为 java.lang.String 数据类型后可被分析。

Servlet 错误处理程序实例

以下是 Servlet 实例，将应对任何您所定义的错误或异常发生时的错误处理程序。

本实例让您对 Servlet 中的异常处理有基本的了解，您可以使用相同的概念编写更复杂的异常处理应用程序：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class ErrorHandler extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 分析 Servlet 异常
      Throwable throwable = (Throwable)
      request.getAttribute("javax.servlet.error.exception");
      Integer statusCode = (Integer)
      request.getAttribute("javax.servlet.error.status_code");
      String servletName = (String)
      request.getAttribute("javax.servlet.error.servlet_name");
      if (servletName == null){
         servletName = "Unknown";
      }
      String requestUri = (String)
      request.getAttribute("javax.servlet.error.request_uri");
      if (requestUri == null){
         requestUri = "Unknown";
      }

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Error/Exception Information";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n");

      if (throwable == null && statusCode == null){
         out.println("<h2>Error information is missing</h2>");
         out.println("Please return to the <a href=\"" +
           response.encodeURL("http://localhost:8080/") +
           "\">Home Page</a>.");
      }else if (statusCode != null){
         out.println("The status code : " + statusCode);
      }else{
         out.println("<h2 class="tutheader">Error information</h2>");
         out.println("Servlet Name : " + servletName +
                             "</br></br>");
         out.println("Exception Type : " +
                             throwable.getClass( ).getName( ) +
                             "</br></br>");
         out.println("The request URI: " + requestUri +
                             "<br><br>");
         out.println("The exception message: " +
                                 throwable.getMessage( ));
      }
      out.println("</body>");
      out.println("</html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

以通常的方式编译 **ErrorHandler.java**，把您的类文件放入<Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

让我们在 web.xml 文件中添加如下配置来处理异常：

```
<servlet>
        <servlet-name>ErrorHandler</servlet-name>
        <servlet-class>ErrorHandler</servlet-class>
</servlet>
<!-- servlet mappings -->
<servlet-mapping>
        <servlet-name>ErrorHandler</servlet-name>
        <url-pattern>/ErrorHandler</url-pattern>
</servlet-mapping>
<error-page>
    <error-code>404</error-code>
    <location>/ErrorHandler</location>
</error-page>
<error-page>
    <exception-type>java.lang.Throwable</exception-type >
    <location>/ErrorHandler</location>
</error-page>
```

现在，尝试使用一个会产生异常的 Servlet，或者输入一个错误的 URL，这将触发 Web 容器调用 **ErrorHandler** 的 Servlet，并显示适当的消息。例如，如果您输入了一个错误的 URL，那么它将显示下面的结果：

```
The status code : 404
```

上面的代码在某些 Web 浏览器中可能无法正常工作。因此，请尽量尝试使用 Mozilla 和 Safari 浏览器，在这两种浏览器中它们应该能够正常工作。

Servlet Cookies 处理

Cookies 是存储在客户端计算机上的文本文件，并保留了各种跟踪信息。Java Servlet 显然支持 HTTP Cookies。

识别返回用户包括三个步骤：

- 服务器脚本向浏览器发送一组 Cookies。例如：姓名、年龄或识别号码等。
- 浏览器将这些信息存储在本地计算机上，以备将来使用。
- 当下一次浏览器向 Web 服务器发送任何请求时，浏览器会把这些 Cookies 信息发送到服务器，服务器将使用这些信息来识别用户。

本章将向您讲解如何设置或重置 Cookies，如何访问它们，以及如何将它们删除。

Cookie 剖析

Cookies 通常设置在 HTTP 头信息中（虽然 JavaScript 也可以直接在浏览器上设置一个 Cookie）。设置 Cookie 的 Servlet 会发送如下的头信息：

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
                 path=/; domain=w3cschool.cc
Connection: close
Content-Type: text/html
```

正如您所看到的，Set-Cookie 头包含了一个名称值对、一个 GMT 日期、一个路径和一个域。名称和值会被 URL 编码。expires 字段是一个指令，告诉浏览器在给定的时间和日期之后"忘记"该 Cookie。

如果浏览器被配置为存储 Cookies，它将会保留此信息直到到期日期。如果用户的浏览器指向任何匹配该 Cookie 的路径和域的页面，它会重新发送 Cookie 到服务器。浏览器的头信息可能如下所示：

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

Servlet 就能够通过请求方法 *request.getCookies()* 访问 Cookie，该方法将返回一个 *Cookie* 对象的数组。

Servlet Cookies 方法

以下是在 Servlet 中操作 Cookies 时可使用的有用的方法列表。

方法	描述
public void setDomain(String pattern)	该方法设置 cookie 适用的域，例如 w3cschool.cc。
public String getDomain()	该方法获取 cookie 适用的域，例如 w3cschool.cc。
public void setMaxAge(int expiry)	该方法设置 cookie 过期的时间（以秒为单位）。如果不这样设置，cookie 只会在当前 session 会话中持续有效。
public int getMaxAge()	该方法返回 cookie 的最大生存周期（以秒为单位），默认情况下，-1 表示 cookie 将持续下去，直到浏览器关闭。
public String getName()	该方法返回 cookie 的名称。名称在创建后不能改变。
public void setValue(String newValue)	该方法设置与 cookie 关联的值。
public String getValue()	该方法获取与 cookie 关联的值。
public void setPath(String uri)	该方法设置 cookie 适用的路径。如果您不指定路径，与当前页面相同目录下的（包括子目录下的）所有 URL 都会返回 cookie。
public String getPath()	该方法获取 cookie 适用的路径。
public void setSecure(boolean flag)	该方法设置布尔值，表示 cookie 是否应该只在加密的（即 SSL）连接上发送。
public void setComment(String purpose)	该方法规定了描述 cookie 目的的注释。该注释在浏览器向用户呈现 cookie 时非常有用。
public String getComment()	该方法返回了描述 cookie 目的的注释，如果 cookie 没有注释则返回 null。

通过 Servlet 设置 Cookies

通过 Servlet 设置 Cookies 包括三个步骤：

(1) 创建一个 **Cookie** 对象：您可以调用带有 cookie 名称和 cookie 值的 Cookie 构造函数，cookie 名称和 cookie 值都是字符串。

```
Cookie cookie = new Cookie("key","value");
```

请记住，无论是名字还是值，都不应该包含空格或以下任何字符：

```
[ ] ( ) = , " / ? @ : ;
```

(2) 设置最大生存周期：您可以使用 setMaxAge 方法来指定 cookie 能够保持有效的时间（以秒为单位）。下面将设置一个最长有效期为 24 小时的 cookie。

```
cookie.setMaxAge(60*60*24);
```

(3) 发送 **Cookie** 到 **HTTP** 响应头：您可以使用 **response.addCookie** 来添加 HTTP 响应头中的 Cookies，如下所示：

```
response.addCookie(cookie);
```

实例

让我们修改我们的 表单数据实例，为名字和姓氏设置 Cookies。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloForm extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 为名字和姓氏创建 Cookies
      Cookie firstName = new Cookie("first_name",
                      request.getParameter("first_name"));
      Cookie lastName = new Cookie("last_name",
                      request.getParameter("last_name"));

      // 为两个 Cookies 设置过期日期为 24 小时后
      firstName.setMaxAge(60*60*24);
      lastName.setMaxAge(60*60*24);

      // 在响应头中添加两个 Cookies
      response.addCookie( firstName );
      response.addCookie( lastName );

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "设置 Cookies 实例";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>名字</b>："
                + request.getParameter("first_name") + "\n" +
                "  <li><b>姓氏</b>："
                + request.getParameter("last_name") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
}
```

编译上面的 Servlet **HelloForm**，并在 web.xml 文件中创建适当的条目，最后尝试下面的 HTML 页面来调用 Servlet。

```
<html>
<body>
<form action="HelloForm" method="GET">
名字：<input type="text" name="first_name">
<br />
姓氏：<input type="text" name="last_name" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

保存上面的 HTML 内容到文件 hello.htm 中，并把它放在 <Tomcat-installation-directory>/webapps/ROOT 目录中。当您访问 *http://localhost:8080/Hello.htm* 时，上面表单的实际输出如下所示：

名字： 姓氏： 提交

尝试输入名字和姓氏，然后点击"提交"按钮，名字和姓氏将显示在屏幕上，同时会设置 firstName 和 lastName 这两个 Cookies，当下次您按下提交按钮时，会被这两个 Cookies 传回到服务器。

下一节会讲解如何在 Web 应用程序中访问这些 Cookies。

通过 Servlet 读取 Cookies

要读取 Cookies，您需要通过调用 *HttpServletRequest* 的 **getCookies( )** 方法创建一个 *javax.servlet.http.Cookie* 对象的数组。然后循环遍历数组，并使用 getName() 和 getValue() 方法来访问每个 cookie 和关联的值。

实例

让我们读取上面的实例中设置的 Cookies

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class ReadCookies extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      Cookie cookie = null;
      Cookie[] cookies = null;
      // 获取与该域相关的 Cookies 的数组
      cookies = request.getCookies();

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Reading Cookies Example";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" );
      if( cookies != null ){
         out.println("<h2>查找 Cookies 名称和值</h2>");
         for (int i = 0; i < cookies.length; i++){
            cookie = cookies[i];
            out.print("名称：" + cookie.getName( ) + ", ");
            out.print("值：" + cookie.getValue( )+" <br/>");
         }
      }else{
          out.println(
            "<h2 class="tutheader">未找到 Cookies</h2>");
      }
      out.println("</body>");
      out.println("</html>");
   }
}
```

编译上面的 Servlet **ReadCookies**，并在 web.xml 文件中创建适当的条目。如果您已经设置了 first*name cookie 为 "John"*，*last_name cookie 为 "Player"*，尝试运行 _*http://localhost:8080/ReadCookies*，将显示如下结果：

```
<tr><td>
<h2>查找 Cookies 名称和值</h2>
名称：first_name，值：John
名称：last_name，值：Player</td></tr>
```

通过 Servlet 删除 Cookies

删除 Cookies 是非常简单的。如果您想删除一个 cookie，那么您只需要按照以下三个步骤进行：

- 读取一个现有的 cookie，并把它存储在 Cookie 对象中。

- 使用 **setMaxAge()** 方法设置 cookie 的年龄为零，来删除现有的 cookie。
- 把这个 cookie 添加到响应头。

实例

下面的例子将删除现有的名为 "first_name" 的 cookie，当您下次运行 ReadCookies 的 Servlet 时，它会返回 first_name 为空值。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class DeleteCookies extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      Cookie cookie = null;
      Cookie[] cookies = null;
      // 获取与该域相关的 Cookies 的数组
      cookies = request.getCookies();

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Delete Cookies Example";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" );
       if( cookies != null ){
         out.println("<h2>Cookies 名称和值</h2>");
         for (int i = 0; i < cookies.length; i++){
            cookie = cookies[i];
            if((cookie.getName( )).compareTo("first_name") == 0 ){
                 cookie.setMaxAge(0);
                 response.addCookie(cookie);
                 out.print("已删除的 cookie：" +
                              cookie.getName( ) + "<br/>");
            }
            out.print("名称：" + cookie.getName( ) + ", ");
            out.print("值：" + cookie.getValue( )+" <br/>");
         }
      }else{
          out.println(
            "<h2 class="tutheader">No cookies founds</h2>");
      }
      out.println("</body>");
      out.println("</html>");
   }
}
```

编译上面的 Servlet **DeleteCookies**，并在 web.xml 文件中创建适当的条目。现在运行 *http://localhost:8080/DeleteCookies*，将显示如下结果：

```
<h2>Cookies 名称和值</h2>
已删除的 cookie：first_name
名称：first_name, 值：John
名称：last_name, 值：Player
```

现在尝试运行 *http://localhost:8080/ReadCookies*，它将只显示一个 cookie，如下所示：

```
<h2>查找 Cookies 名称和值</h2>
名称：last_name, 值：Player
```

您可以手动在 Internet Explorer 中删除 Cookies。在"工具"菜单，选择"Internet 选项"。如果要删除所有的 Cookies，请按"删除 Cookies"。

Servlet Session 跟踪

HTTP 是一种"无状态"协议，这意味着每次客户端检索网页时，客户端打开一个单独的连接到 Web 服务器，服务器会自动不保留之前客户端请求的任何记录。

但是仍然有以下三种方式来维持 Web 客户端和 Web 服务器之间的 session 会话：

Cookies

一个 Web 服务器可以分配一个唯一的 session 会话 ID 作为每个 Web 客户端的 cookie，对于客户端的后续请求可以使用接收到的 cookie 来识别。

这可能不是一个有效的方法，因为很多浏览器不支持 cookie，所以我们建议不要使用这种方式来维持 session 会话。

隐藏的表单字段

一个 Web 服务器可以发送一个隐藏的 HTML 表单字段，以及一个唯一的 session 会话 ID，如下所示：

```
<input type="hidden" name="sessionid" value="12345">
```

该条目意味着，当表单被提交时，指定的名称和值会被自动包含在 GET 或 POST 数据中。每次当 Web 浏览器发送回请求时，session_id 值可以用于保持不同的 Web 浏览器的跟踪。

这可能是一种保持 session 会话跟踪的有效方式，但是点击常规的超文本链接（<A HREF...>）不会导致表单提交，因此隐藏的表单字段也不支持常规的 session 会话跟踪。

URL 重写

您可以在每个 URL 末尾追加一些额外的数据来标识 session 会话，服务器会把该 session 会话标识符与已存储的有关 session 会话的数据相关联。

例如，http://w3cschool.cc/file.htm;sessionid=12345，session 会话标识符被附加为 sessionid=12345，标识符可被 Web 服务器访问以识别客户端。

URL 重写是一种更好的维持 session 会话的方式，它在浏览器不支持 cookie 时能够很好地工作，但是它的缺点是会动态生成每个 URL 来为页面分配一个 session 会话 ID，即使是在很简单的静态 HTML 页面中也会如此。

HttpSession 对象

除了上述的三种方式，Servlet 还提供了 HttpSession 接口，该接口提供了一种跨多个页面请求或访问网站时识别用户以及存储有关用户信息的方式。

Servlet 容器使用这个接口来创建一个 HTTP 客户端和 HTTP 服务器之间的 session 会话。会话持续一个指定的时间段，跨多个连接或页面请求。

您会通过调用 HttpServletRequest 的公共方法 **getSession()** 来获取 HttpSession 对象，如下所示：

```
HttpSession session = request.getSession();
```

你需要在向客户端发送任何文档内容之前调用 *request.getSession()*。下面总结了 HttpSession 对象中可用的几个重要的方法：

方法	描述
public Object getAttribute(String name)	该方法返回在该 session 会话中具有指定名称的对象，如果没有指定名称的对象，则返回 null。
public Enumeration getAttributeNames()	该方法返回 String 对象的枚举，String 对象包含所有绑定到该 session 会话的对象的名称。
public long getCreationTime()	该方法返回该 session 会话被创建的时间，自格林尼治标准时间 1970 年 1 月 1 日午夜算起，以毫秒为单位。
public String getId()	该方法返回一个包含分配给该 session 会话的唯一标识符的字符串。
public long getLastAccessedTime()	该方法返回客户端最后一次发送与该 session 会话相关的请求的时间自格林尼治标准时间 1970 年 1 月 1 日午夜算起，以毫秒为单位。
public int getMaxInactiveInterval()	该方法返回 Servlet 容器在客户端访问时保持 session 会话打开的最大时间间隔，以秒为单位。
public void invalidate()	该方法指示该 session 会话无效，并解除绑定到它上面的任何对象。
public boolean isNew(	如果客户端还不知道该 session 会话，或者如果客户选择不参入该 session 会话，则该方法返回 true。
public void removeAttribute(String name)	该方法将从该 session 会话移除指定名称的对象。
public void setAttribute(String name, Object value)	该方法使用指定的名称绑定一个对象到该 session 会话。
public void setMaxInactiveInterval(int interval)	该方法在 Servlet 容器指示该 session 会话无效之前，指定客户端请求之间的时间，以秒为单位。

Session 跟踪实例

本实例说明了如何使用 HttpSession 对象获取 session 会话创建时间和最后访问时间。如果不存在 session 会话，我们将通过请求创建一个新的 session 会话。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class SessionTrack extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
```

```
        // 如果不存在 session 会话，则创建一个 session 对象
        HttpSession session = request.getSession(true);
        // 获取 session 创建时间
        Date createTime = new Date(session.getCreationTime());
        // 获取该网页的最后一次访问时间
        Date lastAccessTime = 
                          new Date(session.getLastAccessedTime());

        String title = "欢迎回到我的网站";
        Integer visitCount = new Integer(0);
        String visitCountKey = new String("visitCount");
        String userIDKey = new String("userID");
        String userID = new String("ABCD");

        // 检查网页上是否有新的访问者
        if (session.isNew()){
           title = "欢迎来到我的网站";
           session.setAttribute(userIDKey, userID);
        } else {
           visitCount = (Integer)session.getAttribute(visitCountKey);
           visitCount = visitCount + 1;
           userID = (String)session.getAttribute(userIDKey);
        }
        session.setAttribute(visitCountKey,  visitCount);

        // 设置响应内容类型
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String docType =
        "<!doctype html public \"-//w3c//dtd html 4.0 " +
        "transitional//en\">\n";
        out.println(docType +
                  "<html>\n" +
                  "<head><title>" + title + "</title></head>\n" +
                  "<body bgcolor=\"#f0f0f0\">\n" +
                  "<h1 align=\"center\">" + title + "</h1>\n" +
                   "<h2 align=\"center\">Session 信息</h2>\n" +
                  "<table border=\"1\" align=\"center\">\n" +
                  "<tr bgcolor=\"#949494\">\n" +
                  "  <th>Session 信息</th><th>值</th></tr>\n" +
                  "<tr>\n" +
                  "  <td>id</td>\n" +
                  "  <td>" + session.getId() + "</td></tr>\n" +
                  "<tr>\n" +
                  "  <td>Creation Time</td>\n" +
                  "  <td>" + createTime +
                  "  </td></tr>\n" +
                  "<tr>\n" +
                  "  <td>Time of Last Access</td>\n" +
                  "  <td>" + lastAccessTime +
                  "  </td></tr>\n" +
                  "<tr>\n" +
                  "  <td>User ID</td>\n" +
                  "  <td>" + userID +
                  "  </td></tr>\n" +
                  "<tr>\n" +
                  "  <td>Number of visits</td>\n" +
                  "  <td>" + visitCount + "</td></tr>\n" +
                  "</table>\n" +
                  "</body></html>");
    }
}
```

编译上面的 Servlet **SessionTrack**，并在 web.xml 文件中创建适当的条目。在浏览器地址栏输入 *http://localhost:8080/SessionTrack*，当您第一次运行时将显示如下结果：

```
<h1>欢迎来到我的网站</h1>
<h2>Session 信息</h2>
<table>
<tbody>
<tr bgcolor="#949494"><th>Session 信息</th><th>值</th></tr>
<tr><td>id</td><td>0AE3EC93FF44E3C525B4351B77ABB2D5</td></tr>
<tr><td>Creation Time</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>Time of Last Access</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>User ID</td><td>ABCD</td></tr>
<tr><td>Number of visits</td><td>0</td></tr>
</tbody>
</table>
```

再次尝试运行相同的 Servlet，它将显示如下结果：

```
<h1>欢迎回到我的网站</h1>
<h2>Session 信息</h2>
<table>
<tbody>
<tr bgcolor="#949494"><th>Session 信息</th><th>值</th></tr>
<tr><td>id</td><td>0AE3EC93FF44E3C525B4351B77ABB2D5</td></tr>
<tr><td>Creation Time</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>Time of Last Access</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>User ID</td><td>ABCD</td></tr>
<tr><td>Number of visits</td><td>1</td></tr>
</tbody>
</table>
```

删除 Session 会话数据

当您完成了一个用户的 session 会话数据，您有以下几种选择：

- 移除一个特定的属性：您可以调用 *public void removeAttribute(String name)* 方法来删除与特定的键相关联的值。 to delete the value associated with a particular key.
- 删除整个 **session** 会话：您可以调用 *public void invalidate()* 方法来丢弃整个 session 会话。
- 设置 **session** 会话过期时间：您可以调用 *public void setMaxInactiveInterval(int interval)*

方法来单独设置 session 会话超时。

- 注销用户：如果使用的是支持 servlet 2.4 的服务器，您可以调用 **logout** 来注销 Web 服务器的客户端，并把属于所有用户的所有 session 会话设置为无效。
- **web.xml** 配置：如果您使用的是 Tomcat，除了上述方法，您还可以在 web.xml 文件中配置 session 会话超时，如下所示：

```
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
```

上面实例中的超时时间是以分钟为单位，将覆盖 Tomcat 中默认的 30 分钟超时时间。

在一个 Servlet 中的 getMaxInactiveInterval() 方法会返回 session 会话的超时时间，以秒为单位。所以，如果在 web.xml 中配置 session 会话超时时间为 15 分钟，那么 getMaxInactiveInterval() 会返回 900。

Servlet 数据库访问

本教程假定您已经了解了 JDBC 应用程序的工作方式。在您开始学习 Servlet 数据库访问之前，请确保您已经有适当的 JDBC 环境设置和数据库。

从基本概念下手，让我们来创建一个简单的表，并在表中创建几条记录。

创建数据库表

在测试数据库 **TEST** 中创建 **Employees** 表，请按以下步骤进行：

步骤 1：

打开命令行提示符（**Command Prompt**），并更改进入到安装目录，如下所示：

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

步骤 2：

登录到数据库，如下所示：

```
C:\Program Files\MySQL\bin>mysql -u root -p
Enter password: ********
mysql>
```

步骤 3：

在测试数据库 **TEST** 中创建 **Employee** 表，如下所示：

```
mysql> use TEST;
mysql> create table Employees
    (
     id int not null,
     age int not null,
     first varchar (255),
     last varchar (255)
    );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

创建数据记录

最后，在 Employee 表中创建几条记录，如下所示：

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

访问数据库

下面的实例演示了如何使用 Servlet 访问 TEST 数据库。

```
// 加载必需的库
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class DatabaseAccess extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // JDBC 驱动器名称和数据库的 URL
      static final String JDBC_DRIVER="com.mysql.jdbc.Driver";
      static final String DB_URL="jdbc:mysql://localhost/TEST";

      //  数据库的凭据
      static final String USER = "root";
      static final String PASS = "password";

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      String title = "数据库结果";
      String docType =
        "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";
         out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor=\"#f0f0f0\">\n" +
         "<h1 align=\"center\">" + title + "</h1>\n");
      try{
         // 注册 JDBC 驱动器
         Class.forName("com.mysql.jdbc.Driver");

         // 打开一个连接
         conn = DriverManager.getConnection(DB_URL,USER,PASS);

         // 执行 SQL 查询
```

```
         stmt = conn.createStatement();
         String sql;
         sql = "SELECT id, first, last, age FROM Employees";
         ResultSet rs = stmt.executeQuery(sql);

         // 从结果集中提取数据
         while(rs.next()){
            // 根据列名称检索
            int id  = rs.getInt("id");
            int age = rs.getInt("age");
            String first = rs.getString("first");
            String last = rs.getString("last");

            // 显示值
            out.println("ID: " + id + "<br>");
            out.println(", Age: " + age + "<br>");
            out.println(", First: " + first + "<br>");
            out.println(", Last: " + last + "<br>");
         }
         out.println("</body></html>");

         // 清理环境
         rs.close();
         stmt.close();
         conn.close();
      }catch(SQLException se){
         // 处理 JDBC 错误
         se.printStackTrace();
      }catch(Exception e){
         // 处理 Class.forName 错误
         e.printStackTrace();
      }finally{
         // 最后是用于关闭资源的块
         try{
            if(stmt!=null)
               stmt.close();
         }catch(SQLException se2){
         }// 我们不能做什么
         try{
            if(conn!=null)
            conn.close();
         }catch(SQLException se){
            se.printStackTrace();
         }//end finally try
      } //end try
   }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>DatabaseAccess</servlet-name>
     <servlet-class>DatabaseAccess</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>DatabaseAccess</servlet-name>
     <url-pattern>/DatabaseAccess</url-pattern>
 </servlet-mapping>
....
```

现在调用这个 Servlet，输入链接：http://localhost:8080/DatabaseAccess ，将显示以下响应结果：

```
<h1 align="center">数据库结果</h1>
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
```

Servlet 文件上传

Servlet 可以与 HTML form 标签一起使用，来允许用户上传文件到服务器。上传的文件可以是文本文件或图像文件或任何文档。

创建一个文件上传表单

下面的 HTML 代码创建了一个文件上传表单。以下几点需要注意：

- 表单 **method** 属性应该设置为 **POST** 方法，不能使用 GET 方法。
- 表单 **enctype** 属性应该设置为 **multipart/form-data**.
- 表单 **action** 属性应该设置为在后端服务器上处理文件上传的 Servlet 文件。下面的实例使用了 **UploadServlet** Servlet 来上传文件。
- 上传单个文件，您应该使用单个带有属性 type="file" 的 <input .../> 标签。为了允许多个文件上传，请包含多个 name 属性值不同的 input 标签。输入标签具有不同的名称属性的值。浏览器会为每个 input 标签关联一个浏览按钮。

```
<html>
<head>
<title>文件上传表单</title>
</head>
<body>
<h3>文件上传：</h3>
请选择要上传的文件：<br />
<form action="UploadServlet" method="post"
                        enctype="multipart/form-data">
<input type="file" name="file" size="50" />
<br />
<input type="submit" value="上传文件" />
</form>
</body>
</html>
```

这将显示下面的结果，允许用户从本地计算机选择一个文件，当用户点击"上传文件"时，表单会连同从本地计算机选择的文件一起提交：

```
<b>文件上传：</b>
请选择要上传的文件：<br />
<input type="file" name="file" size="50" />
<br />
<input type="button" value="上传文件" />
<br />
注：这只是虚拟的表单，不会正常工作。
```

编写后台 Servlet

以下是 Servlet **UploadServlet**，会接受上传的文件，并把它储存在目录 <Tomcat-installation-directory>/webapps/data 中。这个目录名也可以使用外部配置来添加，比如 web.xml 中的 **context-param** 元素，如下所示：

```
<web-app>
....
<context-param>
    <description>Location to store uploaded file</description>
    <param-name>file-upload</param-name>
    <param-value>
         c:\apache-tomcat-5.5.29\webapps\data\
     </param-value>
</context-param>
....
</web-app>
```

以下是 UploadServlet 的源代码，可以一次处理多个文件的上传。在继续操作之前，请确认下列各项：

- 下面的实例依赖于 FileUpload，所以一定要确保在您的 classpath 中有最新版本的 **commons-fileupload.x.x.jar** 文件。可以从 http://commons.apache.org/fileupload/ 下载。
- FileUpload 依赖于 Commons IO，所以一定要确保在您的 classpath 中有最新版本的 **commons-io-x.x.jar** 文件。可以从 http://commons.apache.org/io/ 下载。
- 在测试下面实例时，您上传的文件大小不能大于 *maxFileSize*，否则文件将无法上传。
- 请确保已经提前创建好目录 c:\temp and c:\apache-tomcat-5.5.29\webapps\data。

```
// 导入必需的 java 库
import java.io.*;
import java.util.*;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.fileupload.FileItem;
import org.apache.commons.fileupload.FileUploadException;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;
import org.apache.commons.io.output.*;

public class UploadServlet extends HttpServlet {

   private boolean isMultipart;
   private String filePath;
   private int maxFileSize = 50 * 1024;
   private int maxMemSize = 4 * 1024;
   private File file ;

   public void init( ){
      // 获取文件将被存储的位置
      filePath =
             getServletContext().getInitParameter("file-upload");
   }
   public void doPost(HttpServletRequest request,
               HttpServletResponse response)
              throws ServletException, java.io.IOException {
```

```
      // 检查我们有一个文件上传请求
      isMultipart = ServletFileUpload.isMultipartContent(request);
      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter( );
      if( !isMultipart ){
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet upload</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<p>No file uploaded</p>");
         out.println("</body>");
         out.println("</html>");
         return;
      }
      DiskFileItemFactory factory = new DiskFileItemFactory();
      // 文件大小的最大值将被存储在内存中
      factory.setSizeThreshold(maxMemSize);
      // Location to save data that is larger than maxMemSize.
      factory.setRepository(new File("c:\\temp"));

      // 创建一个新的文件上传处理程序
      ServletFileUpload upload = new ServletFileUpload(factory);
      // 允许上传的文件大小的最大值
      upload.setSizeMax( maxFileSize );

      try{
      // 解析请求，获取文件项
      List fileItems = upload.parseRequest(request);

      // 处理上传的文件项
      Iterator i = fileItems.iterator();

      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet upload</title>");
      out.println("</head>");
      out.println("<body>");
      while ( i.hasNext () )
      {
         FileItem fi = (FileItem)i.next();
         if ( !fi.isFormField () )
         {
            // 获取上传文件的参数
            String fieldName = fi.getFieldName();
            String fileName = fi.getName();
            String contentType = fi.getContentType();
            boolean isInMemory = fi.isInMemory();
            long sizeInBytes = fi.getSize();
            // 写入文件
            if( fileName.lastIndexOf("\\") >= 0 ){
               file = new File( filePath +
               fileName.substring( fileName.lastIndexOf("\\"))) ;
            }else{
               file = new File( filePath +
               fileName.substring(fileName.lastIndexOf("\\")+1)) ;
            }
            fi.write( file ) ;
            out.println("Uploaded Filename: " + fileName + "<br>");
         }
      }
      out.println("</body>");
      out.println("</html>");
   }catch(Exception ex) {
       System.out.println(ex);
   }
   }
   public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, java.io.IOException {

        throw new ServletException("GET method used with " +
```

```
                getClass( ).getName( )+": POST method required.");
   }
}
```

编译和运行 Servlet

编译上面的 Servlet UploadServlet，并在 web.xml 文件中创建所需的条目，如下所示：

```
<servlet>
   <servlet-name>UploadServlet</servlet-name>
   <servlet-class>UploadServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>UploadServlet</servlet-name>
   <url-pattern>/UploadServlet</url-pattern>
</servlet-mapping>
```

现在尝试使用您在上面创建的 HTML 表单来上传文件。当您在浏览器中访问：http://localhost:8080/UploadFile.htm 时，它会显示下面的结果，这将有助于您从本地计算机上传任何文件。

```
<b>文件上传：</b>
请选择要上传的文件：<br />
<input type="file" name="file" size="50" />
<br />
<input type="button" value="上传文件" />
```

如果您的 Servelt 脚本能正常工作，那么您的文件会被上传到 c:\apache-tomcat-5.5.29\webapps\data\ 目录中。

Servlet 处理日期

使用 Servlet 的最重要的优势之一是，可以使用核心 Java 中的大多数可用的方法。本章将讲解 Java 提供的 **java.util** 包中的 **Date** 类，这个类封装了当前的日期和时间。

Date 类支持两个构造函数。第一个构造函数初始化当前日期和时间的对象。

```
Date( )
```

下面的构造函数接受一个参数，该参数等于 1970 年 1 月 1 日午夜以来经过的毫秒数。

```
Date(long millisec)
```

一旦您有一个可用的 Date 对象，您可以调用下列任意支持的方法来使用日期：

方法	描述
boolean after(Date date)	如果调用的 Date 对象中包含的日期在 date 指定的日期之后，则返回 true，否则返回 false。
boolean before(Date date)	如果调用的 Date 对象中包含的日期在 date 指定的日期之前，则返回 true，否则返回 false。
Object clone( )	重复调用 Date 对象。
int compareTo(Date date)	把调用对象的值与 date 的值进行比较。如果两个值是相等的，则返回 0。如果调用对象在 date 之前，则返回一个负值。如果调用对象在 date 之后，则返回一个正值。
int compareTo(Object obj)	如果 obj 是 Date 类，则操作等同于 compareTo(Date)。否则，它会抛出一个 ClassCastException。
boolean equals(Object date)	如果调用的 Date 对象中包含的时间和日期与 date 指定的相同，则返回 true，否则返回 false。
long getTime( )	返回 1970 年 1 月 1 日以来经过的毫秒数。
int hashCode( )	为调用对象返回哈希代码。
void setTime(long time)	设置 time 指定的时间和日期，这表示从 1970 年 1 月 1 日午夜以来经过的时间（以毫秒为单位）。
String toString( )	转换调用的 Date 对象为一个字符串，并返回结果。

获取当前的日期和时间

在 Java Servlet 中获取当前的日期和时间是非常容易的。您可以使用一个简单的 Date 对象的 *toString()* 方法来输出当前的日期和时间，如下所示：

```
// 导入必需的 java 库
import java.io.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class CurrentDate extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "显示当前的日期和时间";
      Date date = new Date();
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">" + date.toString() + "</h2>\n" +
        "</body></html>");
  }
}
```

现在，让我们来编译上面的 Servlet，并在 web.xml 文件中创建适当的条目，然后通过访问 http://localhost:8080/CurrentDate 来调用该 Servlet。这将会产生如下的结果：

```
<h1>显示当前的日期和时间</h1>

<h2>Mon Jun 21 21:46:49 GMT+04:00 2010</h2>
```

尝试刷新 URL http://localhost:8080/CurrentDate，每隔几秒刷新一次您都会发现显示时间的差异。

日期比较

正如上面所提到的，您可以在 Servlet 中使用所有可用的 Java 方法。如果您需要比较两个日期，以下是方法：

- 您可以使用 getTime() 来获取两个对象自 1970 年 1 月 1 日午夜以来经过的时间（以毫秒为单位），然后对这两个值进行比较。
- 您可以使用方法 before( )、after( ) 和 equals( )。由于一个月里 12 号在 18 号之前，例如，new Date(99, 2, 12).before(new Date (99, 2, 18)) 返回 true。

- 您可以使用 compareTo( ) 方法，该方法由 Comparable 接口定义，由 Date 实现。

使用 SimpleDateFormat 格式化日期

SimpleDateFormat 是一个以语言环境敏感的方式来格式化和解析日期的具体类。SimpleDateFormat 允许您选择任何用户定义的日期时间格式化的模式。

让我们修改上面的实例，如下所示：

```
// 导入必需的 java 库
import java.io.*;
import java.text.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class CurrentDate extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "显示当前的日期和时间";
      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">" + ft.format(dNow) + "</h2>\n" +
        "</body></html>");
  }
}
```

再次编译上面的 Servlet，然后通过访问 http://localhost:8080/CurrentDate 来调用该 Servlet。这将会产生如下的结果：

```
<h1>显示当前的日期和时间</h1>

<h2>Mon 2010.06.21 at 10:06:44 PM GMT+04:00</h2>
```

简单的日期格式的格式代码

使用事件模式字符串来指定时间格式。在这种模式下，所有的 ASCII 字母被保留为模式字母，这些字母定义如下：

字符	描述	实例
G	Era 指示器	AD
y	四位数表示的年	2001
M	一年中的月	July 或 07
d	一月中的第几天	10
h	带有 A.M./P.M. 的小时（1~12）	12
H	一天中的第几小时（0~23）	22
m	一小时中的第几分	30
s	一分中的第几秒	55
S	毫秒	234
E	一周中的星期几	Tuesday
D	一年中的第几天	360
F	所在的周是这个月的第几周	2 (second Wed. in July)
w	一年中的第几周	40
W	一月中的第几周	1
a	A.M./P.M. 标记	PM
k	一天中的第几小时（1~24）	24
K	带有 A.M./P.M. 的小时（0~11）	10
z	时区	Eastern Standard Time
'	Escape for text	Delimiter
"	单引号	`

如需查看可用的处理日期方法的完整列表，您可以参考标准的 Java 文档。

Servlet 网页重定向

当文档移动到新的位置，我们需要向客户端发送这个新位置时，我们需要用到网页重定向。当然，也可能是为了负载均衡，或者只是为了简单的随机，这些情况都有可能用到网页重定向。

重定向请求到另一个网页的最简单的方式是使用 response 对象的 **sendRedirect()** 方法。下面是该方法的定义：将请求重定向到另一页的最简单的方法是，用方法的sendRedirect()的响应对象。以下是这种方法的定义：

```
public void HttpServletResponse.sendRedirect(String location)
throws IOException
```

该方法把响应连同状态码和新的网页位置发送回浏览器。您也可以通过把 setStatus() 和 setHeader() 方法一起使用来达到同样的效果：

```
....
String site = "http://www.newpage.com" ;
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
....
```

实例

本实例显示了 Servlet 如何进行页面重定向到另一个位置：

```
import java.io.*;
import java.sql.Date;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PageRedirect extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      // 要重定向的新位置
      String site = new String("http://www.w3cschool.cc");

      response.setStatus(response.SC_MOVED_TEMPORARILY);
      response.setHeader("Location", site);
    }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>PageRedirect</servlet-name>
     <servlet-class>PageRedirect</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>PageRedirect</servlet-name>
     <url-pattern>/PageRedirect</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/PageRedirect 来调用这个 Servlet。这将把您转到给定的 URL http://www.w3cschool.cc 。

Servlet 点击计数器

网页点击计数器

很多时候，您可能有兴趣知道网站的某个特定页面上的总点击量。使用 Servlet 来计算这些点击量是非常简单的，因为一个 Servlet 的生命周期是由它运行所在的容器控制的。

以下是实现一个简单的基于 Servlet 生命周期的网页点击计数器需要采取的步骤：

- 在 init() 方法中初始化一个全局变量。
- 每次调用 doGet() 或 doPost() 方法时，都增加全局变量。
- 如果需要，您可以使用一个数据库表来存储全局变量的值在 destroy() 中。在下次初始化 Servlet 时，该值可在 init() 方法内被读取。这一步是可选的。
- 如果您只想对一个 session 会话计数一次页面点击，那么请使用 isNew() 方法来检查该 session 会话是否已点击过相同页面。这一步是可选的。
- 您可以通过显示全局计数器的值，来在网站上展示页面的总点击量。这一步是可选的。

在这里，我们假设 Web 容器将无法重新启动。如果是重新启动或 Servlet 被销毁，计数器将被重置。

实例

本实例演示了如何实现一个简单的网页点击计数器：

```
import java.io.*;
import java.sql.Date;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PageHitCounter extends HttpServlet{

  private int hitCount;

  public void init()
  {
     // 重置点击计数器
     hitCount = 0;
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");
      // 该方法在 Servlet 被点击时执行
      // 增加 hitCount
      hitCount++;
      PrintWriter out = response.getWriter();
      String title = "总点击量";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">" + hitCount + "</h2>\n" +
        "</body></html>");

  }
  public void destroy()
  {
      // 这一步是可选的，但是如果需要，您可以把 hitCount 的值写入到数据库
  }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>PageHitCounter</servlet-name>
     <servlet-class>PageHitCounter</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>PageHitCounter</servlet-name>
     <url-pattern>/PageHitCounter</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/PageHitCounter 来调用这个 Servlet。这将会在每次页面刷新时，把计数器的值增加 1，结果如下所示：

```
<h1>总点击量</h1>

<h2>6</h2>
```

网站点击计数器

很多时候，您可能有兴趣知道整个网站的总点击量。在 Servlet 中，这也是非常简单的，我们可以使用过滤器做到这一点。

以下是实现一个简单的基于过滤器生命周期的网站点击计数器需要采取的步骤：

- 在过滤器的 init() 方法中初始化一个全局变量。
- 每次调用 doFilter 方法时，都增加全局变量。
- 如果需要，您可以使用一个数据库表来存储全局变量的值在过滤器的 destroy() 中。在下次初始化过滤器时，该值可在 init() 方法内被读取。这一步是可选的。

在这里，我们假设 Web 容器将无法重新启动。如果是重新启动或 Servlet 被销毁，点击计数器将被重置。

实例

本实例演示了如何实现一个简单的网站点击计数器：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class SiteHitCounter implements Filter{

  private int hitCount;

  public void  init(FilterConfig config)
                    throws ServletException{
     // 重置点击计数器
     hitCount = 0;
  }

  public void  doFilter(ServletRequest request,
              ServletResponse response,
              FilterChain chain)
              throws java.io.IOException, ServletException {

      // 把计数器的值增加 1
      hitCount++;

      // 输出计数器
      System.out.println("网站访问统计："+ hitCount );

      // 把请求传回到过滤器链
      chain.doFilter(request,response);
  }
  public void destroy()
  {
      // 这一步是可选的，但是如果需要，您可以把 hitCount 的值写入到数据库
  }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
<filter>
   <filter-name>SiteHitCounter</filter-name>
   <filter-class>SiteHitCounter</filter-class>
</filter>

<filter-mapping>
   <filter-name>SiteHitCounter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

....
```

现在访问网站的任意页面，比如 http://localhost:8080/ 。这将会在每次任意页面被点击时，把计数器的值增加 1，它会在日志中显示以下消息：

```
网站访问统计： 1
网站访问统计： 2
网站访问统计： 3
网站访问统计： 4
网站访问统计： 5
..................
```

Servlet 自动刷新页面

假设有一个网页，它是显示现场比赛成绩或股票市场状况或货币兑换率。对于所有这些类型的页面，您需要定期刷新网页。

Java Servlet 提供了一个机制，使得网页会在给定的时间间隔自动刷新。

刷新网页的最简单的方式是使用响应对象的方法 **setIntHeader()**。以下是这种方法的定义：

```
public void setIntHeader(String header, int headerValue)
```

此方法把头信息 "Refresh" 连同一个表示时间间隔的整数值（以秒为单位）发送回浏览器。

自动刷新页面实例

本实例演示了 Servlet 如何使用 **setIntHeader()** 方法来设置 **Refresh** 头信息，从而实现自动刷新页面。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class Refresh extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置刷新自动加载的事件间隔为 5 秒
      response.setIntHeader("Refresh", 5);

      // 设置响应内容类型
      response.setContentType("text/html");

      // 获取当前的时间
      Calendar calendar = new GregorianCalendar();
      String am_pm;
      int hour = calendar.get(Calendar.HOUR);
      int minute = calendar.get(Calendar.MINUTE);
      int second = calendar.get(Calendar.SECOND);
      if(calendar.get(Calendar.AM_PM) == 0)
        am_pm = "AM";
      else
        am_pm = "PM";

      String CT = hour+":"+ minute +":"+ second +" "+ am_pm;

      PrintWriter out = response.getWriter();
      String title = "使用 Servlet 自动刷新页面";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n"+
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<p>当前时间是：" + CT + "</p>\n");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>Refresh</servlet-name>
     <servlet-class>Refresh</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>Refresh</servlet-name>
     <url-pattern>/Refresh</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/Refresh 来调用这个 Servlet。这将会每隔 5 秒钟显示一次当前系统时间。运行该 Servlet，并等待查看结果：

```
<h1>使用 Servlet 自动刷新页面</h1>

当前时间是：9:44:50 PM
```

Servlet 发送电子邮件

使用 Servlet 发送一封电子邮件是很简单的，但首先您必须在您的计算机上安装 **JavaMail API** 和 **Java Activation Framework**）**JAF**）。

- 您可以从 Java 标准网站下载最新版本的 JavaMail（版本 1.2）。
- 您可以从 Java 标准网站下载最新版本的 JAF（版本 1.1.1）。

下载并解压缩这些文件，在新创建的顶层目录中，您会发现这两个应用程序的一些 jar 文件。您需要把 **mail.jar** 和 **activation.jar** 文件添加到您的 CLASSPATH 中。

发送一封简单的电子邮件

下面的实例将从您的计算机上发送一封简单的电子邮件。这里假设您的本地主机已连接到互联网，并支持发送电子邮件。同时确保 Java Email API 包和 JAF 包的所有的 jar 文件在 CLASSPATH 中都是可用的。

```
// 文件名 SendEmail.java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 收件人的电子邮件 ID
      String to = "abcd@gmail.com";

      // 发件人的电子邮件 ID
      String from = "web@gmail.com";

      // 假设您是从本地主机发送电子邮件
      String host = "localhost";

      // 获取系统的属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象
      Session session = Session.getDefaultInstance(properties);

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      try{
         // 创建一个默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);
         // 设置 From: header field of the header.
         message.setFrom(new InternetAddress(from));
         // 设置 To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                                  new InternetAddress(to));
         // 设置 Subject: header field
         message.setSubject("This is the Subject Line!");
         // 现在设置实际消息
         message.setText("This is actual message");
         // 发送消息
         Transport.send(message);
         String title = "发送电子邮件";
         String res = "成功发送消息...";
         String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";
         out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor=\"#f0f0f0\">\n" +
         "<h1 align=\"center\">" + title + "</h1>\n" +
         "<p align=\"center\">" + res + "</p>\n" +
         "</body></html>");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>SendEmail</servlet-name>
     <servlet-class>SendEmail</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>SendEmail</servlet-name>
     <url-pattern>/SendEmail</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/SendEmail 来调用这个 Servlet。这将会发送一封电子邮件到给定的电子邮件 ID *abcd@gmail.com*，并将显示下面所示的响应：

```
<h1>发送电子邮件</h1>

成功发送消息...
```

如果您想要发送一封电子邮件给多个收件人，那么请使用下面的方法来指定多个电子邮件 ID：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

下面是对参数的描述：

- **type**：这将被设置为 TO、CC 或 BCC。在这里，CC 代表抄送，BCC 代表密件抄送。例如 *Message.RecipientType.TO*。
- **addresses**：这是电子邮件 ID 的数组。当指定电子邮件 ID 时，您需要使用 InternetAddress() 方法。

发送一封 HTML 电子邮件

下面的实例将从您的计算机上发送一封 HTML 格式的电子邮件。这里假设您的本地主机已连接到互联网，并支持发送电子邮件。同时确保 Java Email API 包和 JAF 包的所有的 jar 文件在 CLASSPATH 中都是可用的。

本实例与上一个实例很类似，但是这里我们使用 setContent() 方法来设置第二个参数为 "text/html" 的内容，该参数用来指定 HTML 内容是包含在消息中的。

使用这个实例，您可以发送内容大小不限的 HTML 内容。

```
// 文件名 SendEmail.java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 收件人的电子邮件 ID
      String to = "abcd@gmail.com";

      // 发件人的电子邮件 ID
      String from = "web@gmail.com";

      // 假设您是从本地主机发送电子邮件
      String host = "localhost";

      // 获取系统的属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象
      Session session = Session.getDefaultInstance(properties);

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      try{
         // 创建一个默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);
         // 设置 From: header field of the header.
         message.setFrom(new InternetAddress(from));
         // 设置 To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                                  new InternetAddress(to));
         // 设置 Subject: header field
         message.setSubject("This is the Subject Line!");

         // 设置实际的 HTML 消息，内容大小不限
         message.setContent("<h1>This is actual message</h1>",
                            "text/html" );
         // 发送消息
         Transport.send(message);
         String title = "发送电子邮件";
         String res = "成功发送消息...";
         String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";
         out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor=\"#f0f0f0\">\n" +
         "<h1 align=\"center\">" + title + "</h1>\n" +
         "<p align=\"center\">" + res + "</p>\n" +
         "</body></html>");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

编译并运行上面的 Servlet，在给定的电子邮件 ID 上发送 HTML 消息。

在电子邮件中发送附件

下面的实例将从您的计算机上发送一封带有附件的电子邮件。这里假设您的本地主机已连接到互联网，并支持发送电子邮件。同时确保 Java Email API 包和 JAF 包的所有的 jar 文件在 CLASSPATH 中都是可用的。

```
// 文件名 SendEmail.java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 收件人的电子邮件 ID
      String to = "abcd@gmail.com";

      // 发件人的电子邮件 ID
      String from = "web@gmail.com";

      // 假设您是从本地主机发送电子邮件
      String host = "localhost";

      // 获取系统的属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象
      Session session = Session.getDefaultInstance(properties);

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

       try{
         // 创建一个默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);

         // 设置 From: header field of the header.
         message.setFrom(new InternetAddress(from));

         // 设置 To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // 设置 Subject: header field
         message.setSubject("This is the Subject Line!");

         // 创建消息部分
         BodyPart messageBodyPart = new MimeBodyPart();

         // 填写消息
         messageBodyPart.setText("This is message body");
```

```
            // 创建一个多部分消息
            Multipart multipart = new MimeMultipart();

            // 设置文本消息部分
            multipart.addBodyPart(messageBodyPart);

            // 第二部分是附件
            messageBodyPart = new MimeBodyPart();
            String filename = "file.txt";
            DataSource source = new FileDataSource(filename);
            messageBodyPart.setDataHandler(new DataHandler(source));
            messageBodyPart.setFileName(filename);
            multipart.addBodyPart(messageBodyPart);

            // 发送完整的消息部分
            message.setContent(multipart );

            // 发送消息
            Transport.send(message);
            String title = "发送电子邮件";
            String res = "成功发送电子邮件...";
            String docType =
            "<!doctype html public \"-//w3c//dtd html 4.0 " +
            "transitional//en\">\n";
            out.println(docType +
            "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor=\"#f0f0f0\">\n" +
            "<h1 align=\"center\">" + title + "</h1>\n" +
            "<p align=\"center\">" + res + "</p>\n" +
            "</body></html>");
         }catch (MessagingException mex) {
            mex.printStackTrace();
         }
      }
   }
```

编译并运行上面的 Servlet，在给定的电子邮件 ID 上发送带有文件附件的消息。

用户身份认证部分

如果需要向电子邮件服务器提供用户 ID 和密码进行身份认证，那么您可以设置如下属性：

```
props.setProperty("mail.user", "myuser");
props.setProperty("mail.password", "mypwd");
```

电子邮件发送机制的其余部分与上面讲解的保持一致。

Servlet 包

涉及到 WEB-INF 子目录的 Web 应用程序结构是所有的 Java web 应用程序的标准，并由 Servlet API 规范指定。给定一个顶级目录名 myapp，目录结构如下所示：

```
/myapp
    /images
    /WEB-INF
        /classes
        /lib
```

WEB-INF 子目录中包含应用程序的部署描述符，名为 web.xml。所有的 HTML 文件都位于顶级目录 *myapp* 下。对于 admin 用户，您会发现 ROOT 目录是 myApp 的父目录。

创建包中的 Servlet

WEB-INF/classes 目录包含了所有的 Servlet 类和其他类文件，类文件所在的目录结构与他们的包名称匹配。例如，如果您有一个完全合格的类名称 **com.myorg.MyServlet**，那么这个 Servlet 类必须位于以下目录中：

```
/myapp/WEB-INF/classes/com/myorg/MyServlet.class
```

下面的例子创建包名为 *com.myorg* 的 MyServlet 类。

```
// 为包命名
package com.myorg;

// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {

  private String message;

  public void init() throws ServletException
  {
      // 执行必需的的初始化
      message = "Hello World";
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      // 实际的逻辑是在这里
      PrintWriter out = response.getWriter();
      out.println("<h1>" + message + "</h1>");
  }

  public void destroy()
  {
      // 什么也不做
  }
}
```

编译包中的 Servlet

编译包中的类与编译其他的类没有什么大的不同。最简单的方法是让您的 java 文件保留完全限定路径，如上面提到的类，将被保留在 com.myorg 中。您还需要在 CLASSPATH 中添加该目录。

假设您的环境已正确设置，进入 **<Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes** 目录，并编译 MyServlet.java，如下所示：

```
$ javac MyServlet.java
```

如果 Servlet 依赖于其他库，那么您必须在 CLASSPATH 中也要引用那些 JAR 文件。这里我只引用了 servlet-api.jar JAR 文件，因为我在 Hello World 程序中并没有使用任何其他库。

该命令行使用内置的 javac 编译器，它是 Sun Microsystems Java 软件开发工具包（JDK，全称 Java Software Development Kit）附带的。 Microsystems的Java软件开发工具包（JDK）。为了让该命令正常工作，必须包括您在 PATH 环境变量中所使用的 Java SDK 的位置。

如果一切顺利，上述编译会在同一目录下生成 **MyServlet.class** 文件。下一节将解释如何部把一个已编译的 Servlet 部署到生产中。

Servlet 打包部署

默认情况下，Servlet 应用程序位于路径 <Tomcat-installation-directory>/webapps/ROOT 下，且类文件放在 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

如果您有一个完全合格的类名称 **com.myorg.MyServlet**，那么这个 Servlet 类必须位于 WEB-INF/classes/com/myorg/MyServlet.class 中，您需要在位于 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/ 的 web.xml 文件中创建以下条目：

```
<servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>com.myorg.MyServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>/MyServlet</url-pattern>
</servlet-mapping>
```

上面的条目要被创建在 web.xml 文件中的 <web-app>...</web-app> 标签内。在该文件中可能已经有各种可用的条目，但不要在意。

到这里，您基本上已经完成了，现在让我们使用 <Tomcat-installation-directory>\bin\startup.bat（在 Windows 上）或 <Tomcat-installation-directory>/bin/startup.sh（在 Linux/Solaris 等上）启动 tomcat 服务器，最后在浏览器的地址栏中输入 **http://localhost:8080/MyServlet**。如果一切顺利，您会看到下面的结果：

```
<h1>Hello World</h1>
```

Servlet 调试

测试/调试 Servlet 始终是开发使用过程中的难点。Servlet 往往涉及大量的客户端/服务器交互，可能会出现错误但又难以重现。

这里有一些提示和建议，可以帮助您调试。

System.out.println()

System.out.println() 是作为一个标记来使用的，用来测试一段特定的代码是否被执行。我们也可以打印出变量的值。此外：

- 由于 System 对象是核心 Java 对象的一部分，它可以在不需要安装任何额外类的情况下被用于任何地方。这包括 Servlet、JSP、RMI、EJB's、普通的 Beans 和类，以及独立的应用程序。
- 与在断点处停止不同，写入到 System.out 不会干扰到应用程序的正常执行流程，这使得它在时序是至关重要的时候显得尤为有价值。

下面是使用 System.out.println() 的语法：

```
System.out.println("Debugging message");
```

通过上面的语法生成的所有消息将被记录在 Web 服务器日志文件中。

消息日志

使用适当的日志记录方法来记录所有调试、警告和错误消息，这是非常好的想法，推荐使用 log4J 来记录所有的消息。

Servlet API 还提供了一个简单的输出信息的方式，使用 log() 方法，如下所示：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ContextLog extends HttpServlet {
  public void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
         java.io.IOException {

      String par = request.getParameter("par1");
      // 调用两个 ServletContext.log 方法
      ServletContext context = getServletContext( );

      if (par == null || par.equals(""))
      // 通过 Throwable 参数记录版本
      context.log("No message received:",
          new IllegalStateException("Missing parameter"));
      else
          context.log("Here is the visitor's message: " + par);

      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter( );
      String title = "Context Log";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">Messages sent</h2>\n" +
        "</body></html>");
    } //doGet
}
```

ServletContext 把它的文本消息记录到 Servlet 容器的日志文件中。对于 Tomcat，这些日志可以在 <Tomcat-installation-directory>/logs 目录中找到。

这些日志文件确实对新出现的错误或问题的频率给出指示。正因为如此，建议在通常不会发生的异常的 catch 子句中使用 log() 函数。

使用 JDB 调试器

您可以使用调试 applet 或应用程序的 jdb 命令来调试 Servlet。

为了调试一个 Servlet，我们可以调试 sun.servlet.http.HttpServer，然后把它看成是 HttpServer 执行 Servlet 来响应浏览器端的 HTTP 请求。这与调试 applet 小程序非常相似。与调试 applet 不同的是，实际被调试的程序是 sun.applet.AppletViewer。

大多数调试器会自动隐藏如何调试 applet 的细节。同样的，对于 servlet，您必须帮调试器执行以下操作：

- 设置您的调试器的类路径 classpath，以便它可以找到 sun.servlet.http.Http-Server 和相关的类。
- 设置您的调试器的类路径 classpath，以便它可以找到您的 servlet 和支持的类，通常是

在 server_root/servlets 和 server_root/classes 中。

您通常不会希望 server_root/servlets 在您的 classpath 中，因为它会禁用 servlet 的重新加载。但是这种包含规则对于调试是非常有用的。它允许您的调试器在 HttpServer 中的自定义 Servlet 加载器加载 Servlet 之前在 Servlet 中设置断点。

如果您已经设置了正确的类路径 classpath，就可以开始调试 sun.servlet.http.HttpServer。可以在您想要调试的 Servlet 代码中设置断点，然后通过 Web 浏览器使用给定的 Servlet（http://localhost:8080/servlet/ServletToDebug）向 HttpServer 发出请求。您会看到程序执行到断点处会停止。

使用注释

代码中的注释有助于以各种方式进行调试。注释可用于调试过程的很多其他方式中。

该 Servlet 使用 Java 注释和单行注释（//...），多行注释（/ .../）可用于暂时移除部分 Java 代码。如果 bug 消失，仔细看看您刚才注释的代码并找出问题所在。

客户端和服务器端头信息

有时，当一个 Servlet 并没有像预期那样时，查看原始的 HTTP 请求和响应是非常有用的。如果您熟悉 HTTP 结构，您可以阅读请求和响应，看看这些头信息究竟是什么。

重要的调试技巧

下面列出了一些 Servlet 调试的技巧：

- 请注意，server_root/classes 不会重载，而 server_root/servlets 可能会。
- 要求浏览器显示它所显示的页面的原始内容。这有助于识别格式的问题。它通常是"视图"菜单下的一个选项。
- 通过强制执行完全重新加载页面来确保浏览器还没有缓存前一个请求的输出。在 Netscape Navigator 中，请使用 Shift-Reload，在 Internet Explorer 中，请使用 Shift-Refresh。
- 请确认 servlet 的 init() 方法接受一个 ServletConfig 参数，并调用 super.init(config)。

Servlet 国际化

在我们开始之前，先来看看三个重要术语：

- 国际化（**i18n**）：这意味着一个网站提供了不同版本的翻译成访问者的语言或国籍的内容。
- 本地化（**l10n**）：这意味着向网站添加资源，以使其适应特定的地理或文化区域，例如网站翻译成印地文（Hindi）。
- 区域设置（**locale**）：这是一个特殊的文化或地理区域。它通常指语言符号后跟一个下划线和一个国家符号。例如 "en_US" 表示针对 US 的英语区域设置。

当建立一个全球性的网站时有一些注意事项。本教程不会讲解这些注意事项的完整细节，但它会通过一个很好的实例向您演示如何通过差异化定位（即区域设置）来让网页以不同语言呈现。

Servlet 可以根据请求者的区域设置拾取相应版本的网站，并根据当地的语言、文化和需求提供相应的网站版本。以下是 request 对象中返回 Locale 对象的方法。

```
java.util.Locale request.getLocale()
```

检测区域设置

下面列出了重要的区域设置方法，您可以使用它们来检测请求者的地理位置、语言和区域设置。下面所有的方法都显示了请求者浏览器中设置的国家名称和语言名称。

方法	描述
String getCountry()	该方法以 2 个大写字母形式的 ISO 3166 格式返回该区域设置的国家/地区代码。
String getDisplayCountry()	该方法返回适合向用户显示的区域设置的国家的名称。
String getLanguage()	该方法以小写字母形式的 ISO 639 格式返回该区域设置的语言代码。
String getDisplayLanguage()	该方法返回适合向用户显示的区域设置的语言的名称。
String getISO3Country()	该方法返回该区域设置的国家的三个字母缩写。
String getISO3Language()	该方法返回该区域设置的语言的三个字母的缩写。

实例

本实例演示了如何显示某个请求的语言和相关的国家：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;

public class GetLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 获取客户端的区域设置
      Locale locale = request.getLocale();
      String language = locale.getLanguage();
      String country = locale.getCountry();

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      String title = "检测区域设置";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + language + "</h1>\n" +
        "<h2 align=\"center\">" + country + "</h2>\n" +
        "</body></html>");
  }
}
```

语言设置

Servlet 可以输出以西欧语言（如英语、西班牙语、德语、法语、意大利语、荷兰语等）编写的页面。在这里，为了能正确显示所有的字符，设置 Content-Language 头是非常重要的。

第二点是使用 HTML 实体显示所有的特殊字符，例如，"ñ" 表示 "?"，"¡" 表示 "?"，如下所示：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;

public class DisplaySpanish extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 设置西班牙语言代码
    response.setHeader("Content-Language", "es");

    String title = "En Espa&ntilde;ol";
    String docType =
     "<!doctype html public \"-//w3c//dtd html 4.0 " +
     "transitional//en\">\n";
     out.println(docType +
     "<html>\n" +
     "<head><title>" + title + "</title></head>\n" +
     "<body bgcolor=\"#f0f0f0\">\n" +
     "<h1>" + "En Espa&ntilde;ol:" + "</h1>\n" +
     "<h1>" + "&iexcl;Hola Mundo!" + "</h1>\n" +
     "</body></html>");
  }
}
```

特定于区域设置的日期

您可以使用 java.text.DateFormat 类及其静态方法 getDateTimeInstance() 来格式化特定于区域设置的日期和时间。下面的实例演示了如何格式化特定于某个给定的区域设置的日期：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;
import java.text.DateFormat;
import java.util.Date;

public class DateLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 获取客户端的区域设置
    Locale locale = request.getLocale( );
    String date = DateFormat.getDateTimeInstance(
                                  DateFormat.FULL,
                                  DateFormat.SHORT,
                                  locale).format(new Date( ));

    String title = "特定于区域设置的日期";
    String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
      "<html>\n" +
      "<head><title>" + title + "</title></head>\n" +
      "<body bgcolor=\"#f0f0f0\">\n" +
      "<h1 align=\"center\">" + date + "</h1>\n" +
      "</body></html>");
  }
}
```

特定于区域设置的货币

您可以使用 java.text.NumberFormat 类及其静态方法 getCurrencyInstance() 来格式化数字（比如 long 类型或 double 类型）为特定于区域设置的货币。下面的实例演示了如何格式化特定于某个给定的区域设置的货币：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;
import java.text.NumberFormat;
import java.util.Date;

public class CurrencyLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 获取客户端的区域设置
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getCurrencyInstance(locale);
    String formattedCurr = nft.format(1000000);

    String title = "特定于区域设置的货币";
    String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
      "<html>\n" +
      "<head><title>" + title + "</title></head>\n" +
      "<body bgcolor=\"#f0f0f0\">\n" +
      "<h1 align=\"center\">" + formattedCurr + "</h1>\n" +
      "</body></html>");
  }
}
```

特定于区域设置的百分比

您可以使用 java.text.NumberFormat 类及其静态方法 getPercentInstance() 来格式化特定于区域设置的百分比。下面的实例演示了如何格式化特定于某个给定的区域设置的百分比：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;
import java.text.NumberFormat;
import java.util.Date;

public class PercentageLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 获取客户端的区域设置
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getPercentInstance(locale);
    String formattedPerc = nft.format(0.51);

    String title = "特定于区域设置的百分比";
    String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
      "<html>\n" +
      "<head><title>" + title + "</title></head>\n" +
      "<body bgcolor=\"#f0f0f0\">\n" +
      "<h1 align=\"center\">" + formattedPerc + "</h1>\n" +
      "</body></html>");
  }
}
```

W3School JSP 教程

来源：JSP教程

整理：飞龙

JSP 基础

JSP 简介

什么是Java Server Pages?

JSP全称Java Server Pages，是一种动态网页开发技术。它使用JSP标签在HTML网页中插入Java代码。标签通常以<%开头以%>结束。

JSP是一种Java servlet，主要用于实现Java web应用程序的用户界面部分。网页开发者们通过结合HTML代码、XHTML代码、XML元素以及嵌入JSP操作和命令来编写JSP。

JSP通过网页表单获取用户输入数据、访问数据库及其他数据源，然后动态地创建网页。

JSP标签有多种功能，比如访问数据库、记录用户选择信息、访问JavaBeans组件等，还可以在不同的网页中传递控制信息和共享信息。

为什么使用JSP？

JSP程序与CGI程序有着相似的功能，但和CGI程序相比，JSP程序有如下优势：

- 性能更加优越，因为JSP可以直接在HTML网页中动态嵌入元素而不需要单独引用CGI文件。
- 服务器调用的是已经编译好的JSP文件，而不像CGI/Perl那样必须先载入解释器和目标脚本。
- JSP基于Java Servlets API，因此，JSP拥有各种强大的企业级Java API，包括JDBC，JNDI，EJB，JAXP等等。
- JSP页面可以与处理业务逻辑的servlets一起使用，这种模式被Java servlet 模板引擎所支持。

最后，JSP是Java EE不可或缺的一部分，是一个完整的企业级应用平台。这意味着JSP可以用最简单的方式来实现最复杂的应用。

JSP的优势

以下列出了使用JSP带来的其他好处：

- 与ASP相比：JSP有两大优势。首先，动态部分用Java编写，而不是VB或其他MS专用语言，所以更加强大与易用。第二点就是JSP易于移植到非MS平台上。
- 与纯 Servlets相比：JSP可以很方便的编写或者修改HTML网页而不用去面对大量的println语句。
- 与SSI相比：SSI无法使用表单数据、无法进行数据库链接。

- 与JavaScript相比：虽然JavaScript可以在客户端动态生成HTML，但是很难与服务器交互，因此不能提供复杂的服务，比如访问数据库和图像处理等等。
- 与静态HTML相比：静态HTML不包含动态信息。

接下来呢？

我们将会带您一步一步地来搭建JSP运行环境，这需要有一定的Java基础。

如果您还未学过Java，可以先学习我们为您提供的Java教程。

JSP 开发环境搭建

JSP开发环境是您用来开发、测试和运行JSP程序的地方。

本节将会带您搭建JSP开发环境，具体包括以下几个步骤。

配置Java开发工具（JDK）

这一步涉及Java SDK的下载和PATH环境变量的配置。

您可以从Oracle公司的Java页面中下载SDK：Java SE Downloads

Java SDK下载完后，请按照给定的指示来安装和配置SDK。最后，通过设置PATH和JAVA_HOME环境变量来指明包括java和javac的文件夹路径，通常是java_install_dir/bin和java_install_dir。

假如您用的是Windows系统并且SDK的安装目录为C::\jdk1.5.0_20，那么您就需要在C:\autoexec.bat 文件中添加以下两行：

```
set PATH=C:\jdk1.5.0_20\bin;%PATH%
set JAVA_HOME=C:\jdk1.5.0_20
```

或者，在Windows NT/2000/XP下，您可以直接右击我的电脑图标，选择属性，然后高级，然后环境变量，接下来您就可以很方便地设置PATH变量并且确定退出就行了。

在Linux/Unix系统下，如果SDK的安装目录为/usr/local/jdk1.5.0_20并且使用的是C shell，那么您就需要在.cshrc文件中添加以下两行：

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

或者，假如您正在使用类似于Borland JBuilder、Eclipse、IntelliJ IDEA和Sun ONE Studio这样的集成开发环境，可以试着编译并运行一个简单的程序来确定IDE（集成开发环境）是否已经知道 SDK的安装目录。

本步骤你也可以参考本站Java开发环境配置章节的教程。

设置Web服务器：Tomcat

目前，市场上有很多支持JSP和Servlets开发的Web服务器。他们中的一些可以免费下载和使用，Tomcat就是其中之一。

Apache Tomcat是一个开源软件，可作为独立的服务器来运行JSP和Servlets，也可以集成在Apache Web Server中。以下是Tomcat的配置方法：

- 下载最新版本的Tomcat：http://tomcat.apache.org/。
- 下载完安装文件后，将压缩文件解压到一个方便的地方，比如Windows下的C:\apache-tomcat-5.5.29目录或者Linux/Unix下的/usr/local/apache-tomcat-5.5.29目录，然后创建CATALINA_HOME环境变量指向这些目录。

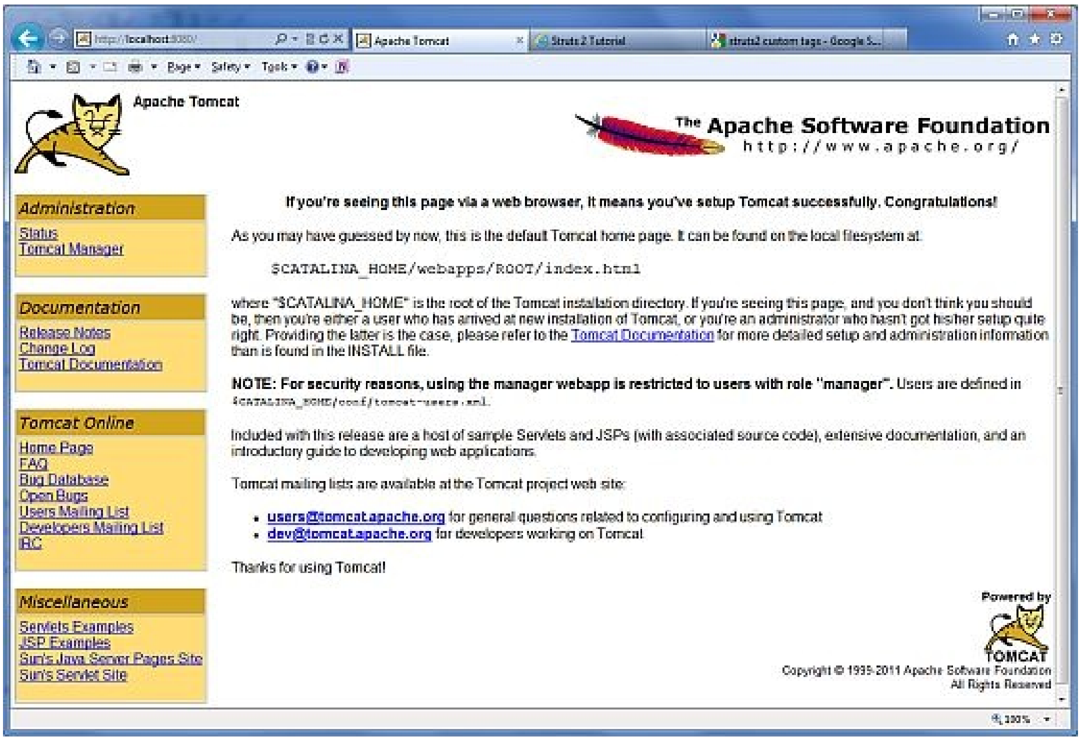
在Windows机器下，Tomcat可以通过执行以下命令来启动：

```
%CATALINA_HOME%\bin\startup.bat
或者
C:\apache-tomcat-5.5.29\bin\startup.bat
```

在Linux/Unix机器下，Tomcat可以通过执行以下命令来启动：

```
$CATALINA_HOME/bin/startup.sh
或者
/usr/local/apache-tomcat-5.5.29/bin/startup.sh
```

成功启动Tomcat后，通过访问http://localhost:8080/便可以使用Tomcat自带的一些web应用了。假如一切顺利的话，您应该能够看到以下的页面：



更多关于配置和运行Tomcat的信息可以在Tomcat提供的文档中找到，或者去Tomcat官网查阅：http://tomcat.apache.org。

在Windows机器下，Tomcat可以通过执行以下命令来停止：

```
%CATALINA_HOME%\bin\shutdown
或者
C:\apache-tomcat-5.5.29\bin\shutdown
```

在Linux/Unix机器下，Tomcat可以通过执行以下命令来停止：

```
$CATALINA_HOME/bin/shutdown.sh
或者
/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

设置CLASSPATH环境变量

由于servlets不是Java SE的一部分，所以您必须标示出servlet类的编译器。

假如您用的是Windows机器，您需要在C:\autoexec.bat文件中添加以下两行：

```
set CATALINA=C:\apache-tomcat-5.5.29
set CLASSPATH=%CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%
```

或者，在Windows NT/2000/XP下，您只要右击我的电脑，选择属性，然后点击高级，然后点击环境变量，接下来便可以设置CLASSPATH变量并且确定退出即可。

在Linux/Unix机器下，假如您使用的是C shell，那么您就需要在.cshrc文件中添加以下两行：

```
setenv CATALINA=/usr/local/apache-tomcat-5.5.29
setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH
```

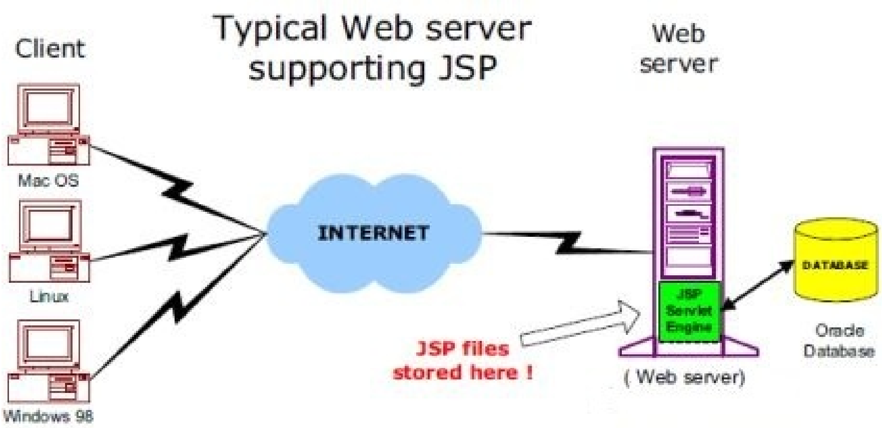
注意：如果您的开发路径是C:\JSPDev (Windows)或者 /usr/JSPDev (Linux/Unix)，那么您就需要将这些路径添加进CLASSPATH变量中。

JSP 结构

网络服务器需要一个JSP引擎，也就是一个容器来处理JSP页面。容器负责截获对JSP页面的请求。本教程使用内嵌JSP容器的Apache来支持JSP开发。

JSP容器与Web服务器协同合作，为JSP的正常运行提供必要的运行环境和其他服务，并且能够正确识别专属于JSP网页的特殊元素。

下图显示了JSP容器和JSP文件在Web应用中所处的位置。

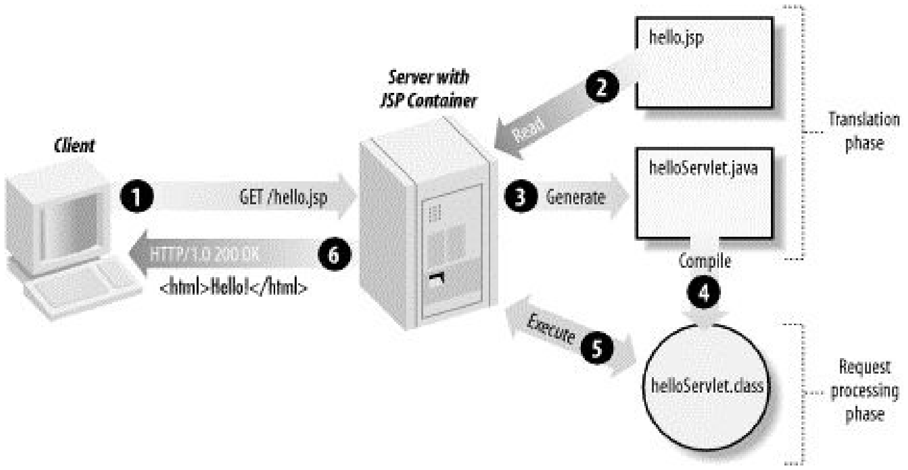


JSP处理

以下步骤表明了Web服务器是如何使用JSP来创建网页的：

- 就像其他普通的网页一样，您的浏览器发送一个HTTP请求给服务器。
- Web服务器识别出这是一个对JSP网页的请求，并且将该请求传递给JSP引擎。通过使用URL或者.jsp文件来完成。
- JSP引擎从磁盘中载入JSP文件，然后将它们转化为servlet。这种转化只是简单地将所有模板文本改用println()语句，并且将所有的JSP元素转化成Java代码。
- JSP引擎将servlet编译成可执行类，并且将原始请求传递给servlet引擎。
- Web服务器的某组件将会调用servlet引擎，然后载入并执行servlet类。在执行过程中，servlet产生HTML格式的输出并将其内嵌于HTTP response中上交给Web服务器。
- Web服务器以静态HTML网页的形式将HTTP response返回到您的浏览器中。
- 最终，Web浏览器处理HTTP response中动态产生的HTML网页，就好像在处理静态网页一样。

以上提及到的步骤可以用下图来表示：



一般情况下，JSP引擎会检查JSP文件对应的servlet是否已经存在，并且检查JSP文件的修改日期是否早于servlet。如果JSP文件的修改日期早于对应的servlet，那么容器就可以确定JSP文件没有被修改过并且servlet有效。这使得整个流程与其他脚本语言（比如PHP）相比要高效快捷一些。

总的来说，JSP网页就是用另一种方式来编写servlet而不用成为Java编程高手。除了解释阶段外，JSP网页几乎可以被当成一个普通的servlet来对待。

JSP 生命周期

理解JSP底层功能的关键就是去理解它们所遵守的生命周期。

JSP生命周期就是从创建到销毁的整个过程，类似于servlet生命周期，区别在于JSP生命周期还包括将JSP文件编译成servlet。

以下是JSP生命周期中所走过的几个阶段：

- 编译阶段：

  servlet容器编译servlet源文件，生成servlet类

- 初始化阶段：

  加载与JSP对应的servlet类，创建其实例，并调用它的初始化方法

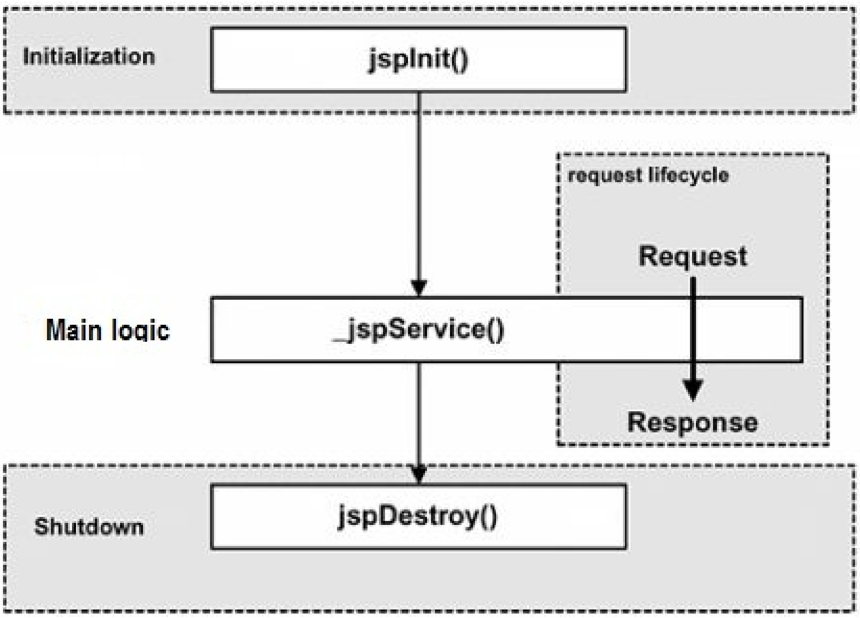
- 执行阶段：

  调用与JSP对应的servlet实例的服务方法

- 销毁阶段：

  调用与JSP对应的servlet实例的销毁方法，然后销毁servlet实例

很明显，JSP生命周期的四个主要阶段和servlet生命周期非常相似，下面给出图示：



JSP编译

当浏览器请求JSP页面时，JSP引擎会首先去检查是否需要编译这个文件。如果这个文件没有被编译过，或者在上次编译后被更改过，则编译这个JSP文件。

编译的过程包括三个步骤：

- 解析JSP文件。
- 将JSP文件转为servlet。
- 编译servlet。

JSP初始化

容器载入JSP文件后，它会在为请求提供任何服务前调用jspInit()方法。如果您需要执行自定义的JSP初始化任务，复写jspInit()方法就行了，就像下面这样：

```
public void jspInit(){
  // 初始化代码
}
```

一般来讲程序只初始化一次，servlet也是如此。通常情况下您可以在jspInit()方法中初始化数据库连接、打开文件和创建查询表。

JSP执行

这一阶段描述了JSP生命周期中一切与请求相关的交互行为，直到被销毁。

当JSP网页完成初始化后，JSP引擎将会调用_jspService()方法。

_jspService()方法需要一个HttpServletRequest对象和一个HttpServletResponse对象作为它的参数，就像下面这样：

```
void _jspService(HttpServletRequest request,
                 HttpServletResponse response)
{
   // 服务端处理代码
}
```

_jspService()方法在每个request中被调用一次并且负责产生与之相对应的response，并且它还负责产生所有7个HTTP方法的回应，比如GET、POST、DELETE等等。

JSP清理

JSP生命周期的销毁阶段描述了当一个JSP网页从容器中被移除时所发生的一切。

jspDestroy()方法在JSP中等价于servlet中的销毁方法。当您需要执行任何清理工作时复写jspDestroy()方法，比如释放数据库连接或者关闭文件夹等等。

jspDestroy()方法的格式如下：

```
public void jspDestroy()
{
   // 清理代码
}
```

实例

JSP生命周期代码实例如下所示：

```
<%@ page contentType="text/html; charset=GB2312" %>
<html><head><title>life.jsp</title></head><body>

<%!
  private int initVar=0;
  private int serviceVar=0;
  private int destroyVar=0;
%>

<%!
  public void jspInit(){
    initVar++;
    System.out.println("jspInit(): JSP被初始化了"+initVar+"次");
  }
  public void jspDestroy(){
    destroyVar++;
    System.out.println("jspDestroy(): JSP被销毁了"+destroyVar+"次");
  }
%>

<%
  serviceVar++;
  System.out.println("_jspService(): JSP共响应了"+serviceVar+"次请求");

  String content1="初始化次数 : "+initVar;
  String content2="响应客户请求次数 : "+serviceVar;
  String content3="销毁次数 : "+destroyVar;
%>

<h1><%=content1 %></h1>
<h1><%=content2 %></h1>
<h1><%=content3 %></h1>

</body></html>
```

JSP 语法

本小节将会简单地介绍一下JSP开发中的基础语法。

脚本程序

脚本程序可以包含任意量的Java语句、变量、方法或表达式，只要它们在脚本语言中是有效的。

脚本程序的语法格式：

```
<% 代码片段 %>
```

或者，您也可以编写与其等价的XML语句，就像下面这样：

```
<jsp:scriptlet>
   代码片段
</jsp:scriptlet>
```

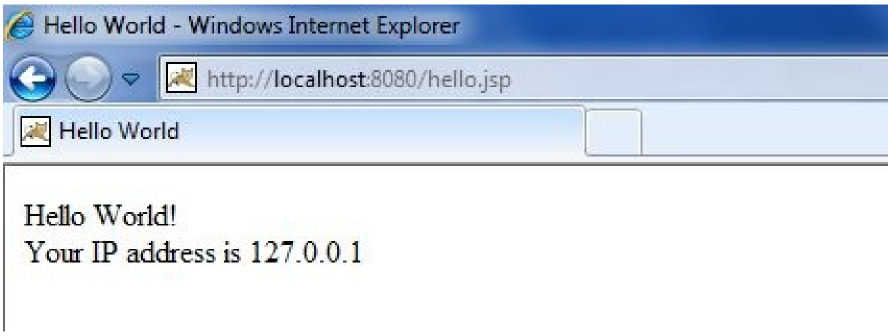
任何文本、HTML标签、JSP元素必须写在脚本程序的外面。

下面给出一个示例，同时也是本教程的第一个JSP示例：

```
<html>
<head><title>Hello World</title></head>
<body>
Hello World!<br/>
<%
out.println("Your IP address is " + request.getRemoteAddr());
%>
</body>
</html>
```

注意：请确保Apache Tomcat已经安装在C:\apache-tomcat-7.0.2目录下并且运行环境已经正确设置。

将以上代码保存在hello.jsp中，然后将它放置在 C:\apache-tomcat-7.0.2\webapps\ROOT目录下，打开浏览器并在地址栏中输入http://localhost:8080/hello.jsp。运行后得到以下结果：



JSP声明

一个声明语句可以声明一个或多个变量、方法，供后面的Java代码使用。在JSP文件中，您必须先声明这些变量和方法然后才能使用它们。

JSP声明的语法格式：

```
<%! declaration; [ declaration; ]+ ... %>
```

或者，您也可以编写与其等价的XML语句，就像下面这样：

```
<jsp:declaration>
   代码片段
</jsp:declaration>
```

程序示例：

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

JSP表达式

一个JSP表达式中包含的脚本语言表达式，先被转化成String，然后插入到表达式出现的地方。

由于表达式的值会被转化成String，所以您可以在一个文本行中使用表达式而不用去管它是否是HTML标签。

表达式元素中可以包含任何符合Java语言规范的表达式，但是不能使用分号来结束表达式。

JSP表达式的语法格式：

```
<%= 表达式 %>
```

同样，您也可以编写与之等价的XML语句：

```
<jsp:expression>
   表达式
</jsp:expression>
```

程序示例：

```
<html>
<head><title>A Comment Test</title></head>
<body>
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
</body>
</html>
```

运行后得到以下结果：

```
Today's date: 11-Sep-2013 21:24:25
```

JSP注释

JSP注释主要有两个作用：为代码作注释以及将某段代码注释掉。

JSP注释的语法格式：

```
<%-- 这里可以填写 JSP 注释 --%>
```

程序示例：

```
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
<%-- 该部分注释在网页中不会被显示--%>
</body>
</html>
```

运行后得到以下结果：

```
A Test of Comments
```

不同情况下使用注释的语法规则：

语法	描述
<%-- 注释 --%>	JSP注释，注释内容不会被发送至浏览器甚至不会被编译
<!-- 注释 -->	HTML注释，通过浏览器查看网页源代码时可以看见注释内容
<\%	代表静态 <%常量
%\>	代表静态 %> 常量
\'	在属性中使用的单引号
\"	在属性中使用的双引号

JSP指令

JSP指令用来设置与整个JSP页面相关的属性。

JSP指令语法格式：

```
<%@ directive attribute="value" %>
```

这里有三种指令标签：

指令	描述
<%@ page ... %>	定义页面的依赖属性，比如脚本语言、error页面、缓存需求等等
<%@ include ... %>	包含其他文件
<%@ taglib ... %>	引入标签库的定义，可以是自定义标签

JSP行为

JSP行为标签使用XML语法结构来控制servlet引擎。它能够动态插入一个文件，重用JavaBean组件，引导用户去另一个页面，为Java插件产生相关的HTML等等。

行为标签只有一种语法格式，它严格遵守XML标准：

```
<jsp:action_name attribute="value" />
```

行为标签基本上是一些预先就定义好的函数，下表罗列出了一些可用的JSP行为标签：：

语法	描述
jsp:include	用于在当前页面中包含静态或动态资源
jsp:useBean	寻找和初始化一个JavaBean组件
jsp:setProperty	设置 JavaBean组件的值
jsp:getProperty	将 JavaBean组件的值插入到 output中
jsp:forward	从一个JSP文件向另一个文件传递一个包含用户请求的request对象
jsp:plugin	用于在生成的HTML页面中包含Applet和JavaBean对象
jsp:element	动态创建一个XML元素
jsp:attribute	定义动态创建的XML元素的属性
jsp:body	定义动态创建的XML元素的主体
jsp:text	用于封装模板数据

JSP隐含对象

JSP支持九个自动定义的变量，江湖人称隐含对象。这九个隐含对象的简介见下表：

对象	描述
request	**HttpServletRequest**类的实例
response	**HttpServletResponse**类的实例
out	**PrintWriter**类的实例，用于把结果输出至网页上
session	**HttpSession**类的实例
application	**ServletContext**类的实例，与应用上下文有关
config	**ServletConfig**类的实例
pageContext	**PageContext**类的实例，提供对JSP页面所有对象以及命名空间的访问
page	类似于Java类中的this关键字
Exception	**Exception**类的对象，代表发生错误的JSP页面中对应的异常对象

控制流语句

JSP提供对Java语言的全面支持。您可以在JSP程序中使用Java API甚至建立Java代码块，包括判断语句和循环语句等等。

判断语句

If...else块，请看下面这个例子：

```
<%! int day = 3; %>
<html>
<head><title>IF...ELSE Example</title></head>
<body>
<% if (day == 1 | day == 7) { %>
      <p> Today is weekend</p>
<% } else { %>
      <p> Today is not weekend</p>
<% } %>
</body>
</html>
```

运行后得到以下结果：

```
Today is not weekend
```

现在来看看switch...case块，与if...else块有很大的不同，它使用out.println()，并且整个都装在脚本程序的标签中，就像下面这样：

```
<%! int day = 3; %>
<html>
<head><title>SWITCH...CASE Example</title></head>
<body>
<%
switch(day) {
case 0:
   out.println("It\'s Sunday.");
   break;
case 1:
   out.println("It\'s Monday.");
   break;
case 2:
   out.println("It\'s Tuesday.");
   break;
case 3:
   out.println("It\'s Wednesday.");
   break;
case 4:
   out.println("It\'s Thursday.");
   break;
case 5:
   out.println("It\'s Friday.");
   break;
default:
   out.println("It's Saturday.");
}
%>
</body>
</html>
```

运行后得出以下结果：

```
It's Wednesday.
```

循环语句

在JSP程序中可以使用Java的三个基本循环类型：for，while，和 do…while。

让我们来看看for循环的例子：

```
<%! int fontSize; %>
<html>
<head><title>FOR LOOP Example</title></head>
<body>
<%for ( fontSize = 1; fontSize <= 3; fontSize++){ %>
   <font color="green" size="<%= fontSize %>">
    JSP Tutorial
   </font><br />
<%}%>
</body>
</html>
```

运行后得到以下结果：

JSP Tutorial
JSP Tutorial
JSP Tutorial

将上例改用while循环来写：

```
<%! int fontSize; %>
<html>
<head><title>WHILE LOOP Example</title></head>
<body>
<%while ( fontSize <= 3){ %>
   <font color="green" size="<%= fontSize %>">
    JSP Tutorial
   </font><br />
<%fontSize++;%>
<%}%>
</body>
</html>
```

运行后得到同样的结果：
JSP Tutorial
JSP Tutorial
JSP Tutorial

JSP运算符

JSP支持所有Java逻辑和算术运算符。

下表罗列出了JSP常见运算符，优先级从高到底：

类别	操作符	结合性
后缀	() [] . (点运算符)	左到右
一元	++ - - ! ~	右到左
可乘性	* / %	左到右
可加性	+ -	左到右
移位	>> >>> <<	左到右
关系	> >= < <=	左到右
相等/不等	== !=	左到右
位与	&	左到右
位异或	^	左到右
位或	|	左到右
逻辑与	&&	左到右
逻辑或	||	左到右
条件判断	?:	右到左
赋值	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

JSP常量

JSP语言定义了以下几个常量：

- Boolean：true and false
- Integer：与Java中的一样
- Floating point：与Java中的一样
- String：以单引号或双引号开始和结束。 " 被转义成 \"，'被转义成 \'， \ 被转义成\
- Null：null

JSP 指令

JSP指令用来设置整个JSP页面相关的属性，如网页的编码方式和脚本语言。

语法格式如下：

```
<%@ directive attribute="value" %>
```

指令可以有很多个属性，它们以键值对的形式存在，并用逗号隔开。

JSP中的三种指令标签：

指令	描述
<%@ page ... %>	定义网页依赖属性，比如脚本语言、error页面、缓存需求等等
<%@ include ... %>	包含其他文件
<%@ taglib ... %>	引入标签库的定义

Page指令

Page指令为容器提供当前页面的使用说明。一个JSP页面可以包含多个page指令。

Page指令的语法格式：

```
<%@ page attribute="value" %>
```

等价的XML格式：

```
<jsp:directive.page attribute="value" />
```

属性

下表列出与Page指令相关的属性：

属性	描述
buffer	指定out对象使用缓冲区的大小
autoFlush	控制out对象的 缓存区
contentType	指定当前JSP页面的MIME类型和字符编码
errorPage	指定当JSP页面发生异常时需要转向的错误处理页面
isErrorPage	指定当前页面是否可以作为另一个JSP页面的错误处理页面
extends	指定servlet从哪一个类继承
import	导入要使用的Java类
info	定义JSP页面的描述信息
isThreadSafe	指定对JSP页面的访问是否为线程安全
language	定义JSP页面所用的脚本语言，默认是Java
session	指定JSP页面是否使用session
isELIgnored	指定是否执行EL表达式
isScriptingEnabled	确定脚本元素能否被使用

Include指令

JSP可以通过include指令来包含其他文件。被包含的文件可以是JSP文件、HTML文件或文本文件。包含的文件就好像是该JSP文件的一部分，会被同时编译执行。

Include指令的语法格式如下：

```
<%@ include file="relative url" %>
```

Include指令中的文件名实际上是一个相对的URL。如果您没有给文件关联一个路径，JSP编译器默认在当前路径下寻找。

等价的XML语法：

```
<jsp:directive.include file="relative url" />
```

Taglib指令

JSP API允许用户自定义标签，一个自定义标签库就是自定义标签的集合。

Taglib指令引入一个自定义标签集合的定义，包括库路径、自定义标签。

Taglib指令的语法：

```
<%@ taglib uri="uri" prefix="prefixOfTag" %>
```

uri属性确定标签库的位置，prefix属性指定标签库的前缀。

等价的XML语法：

```
<jsp:directive.taglib uri="uri" prefix="prefixOfTag" />
```

JSP 动作元素

与JSP指令元素不同的是，JSP动作元素在请求处理阶段起作用。JSP动作元素是用XML语法写成的。

利用JSP动作可以动态地插入文件、重用JavaBean组件、把用户重定向到另外的页面、为Java插件生成HTML代码。

动作元素只有一种语法，它符合XML标准：

```
<jsp:action_name attribute="value" />
```

动作元素基本上都是预定义的函数，JSP规范定义了一系列的标准动作，它用JSP作为前缀，可用的标准动作元素如下：

语法	描述
jsp:include	在页面被请求的时候引入一个文件。
jsp:useBean	寻找或者实例化一个JavaBean。
jsp:setProperty	设置JavaBean的属性。
jsp:getProperty	输出某个JavaBean的属性。
jsp:forward	把请求转到一个新的页面。
jsp:plugin	根据浏览器类型为Java插件生成OBJECT或EMBED标记。
jsp:element	定义动态XML元素
jsp:attribute	设置动态定义的XML元素属性。
jsp:body	设置动态定义的XML元素内容。
jsp:text	在JSP页面和文档中使用写入文本的模板

常见的属性

所有的动作要素都有两个属性：id属性和scope属性。

- **id**属性：

  id属性是动作元素的唯一标识，可以在JSP页面中引用。动作元素创建的id值可以通过PageContext来调用。

- **scope**属性：

该属性用于识别动作元素的生命周期。 id属性和scope属性有直接关系，scope属性定义了相关联id对象的寿命。 scope属性有四个可能的值： (a) page, (b)request, (c)session, 和 (d) application。

<jsp:include>动作元素

<jsp:include>动作元素用来包含静态和动态的文件。该动作把指定文件插入正在生成的页面。语法格式如下：

```
<jsp:include page="relative URL" flush="true" />
```

前面已经介绍过include指令，它是在JSP文件被转换成Servlet的时候引入文件，而这里的jsp:include动作不同，插入文件的时间是在页面被请求的时候。

以下是include动作相关的属性列表。

属性	描述
page	包含在页面中的相对URL地址。
flush	布尔属性，定义在包含资源前是否刷新缓存区。

实例

以下我们定义了两个文件date.jsp和main.jsp，代码如下所示：

date.jsp文件代码：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:include page="date.jsp" flush="true" />
</center>
</body>
</html>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
The include action Example
Today's date: 12-Sep-2013 14:54:22
```

<jsp:useBean>动作元素

jsp:useBean动作用来装载一个将在JSP页面中使用的JavaBean。

这个功能非常有用，因为它使得我们既可以发挥Java组件重用的优势，同时也避免了损失JSP区别于Servlet的方便性。

jsp:useBean动作最简单的语法为：

```
<jsp:useBean id="name" class="package.class" />
```

在类载入后，我们既可以通过 jsp:setProperty 和 jsp:getProperty 动作来修改和检索bean的属性。

以下是useBean动作相关的属性列表。

属性	描述
class	指定Bean的完整包名。
type	指定将引用该对象变量的类型。
beanName	通过 java.beans.Beans 的 instantiate() 方法指定Bean的名字。

在给出具体实例前，让我们先来看下 jsp:setProperty 和 jsp:getProperty 动作元素：

<jsp:setProperty>动作元素

jsp:setProperty用来设置已经实例化的Bean对象的属性，有两种用法。首先，你可以在jsp:useBean元素的外面（后面）使用jsp:setProperty，如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName" property="someProperty" .../>
```

此时，不管jsp:useBean是找到了一个现有的Bean，还是新创建了一个Bean实例，jsp:setProperty都会执行。第二种用法是把jsp:setProperty放入jsp:useBean元素的内部，如下所示：

```
<jsp:useBean id="myName" ... >
...
   <jsp:setProperty name="myName" property="someProperty" .../>
</jsp:useBean>
```

此时，jsp:setProperty只有在新建Bean实例时才会执行，如果是使用现有实例则不执行jsp:setProperty。

jsp:setproperty动作有下面四个属性,如下表：

属性	描述
name	name属性是必需的。它表示要设置属性的是哪个Bean。
property	property属性是必需的。它表示要设置哪个属性。有一个特殊用法：如果property的值是"*"，表示所有名字和Bean属性名字匹配的请求参数都将被传递给相应的属性set方法。
value	value 属性是可选的。该属性用来指定Bean属性的值。字符串数据会在目标类中通过标准的valueOf方法自动转换成数字、boolean、Boolean、 byte、Byte、char、Character。例如，boolean和Boolean类型的属性值（比如"true"）通过 Boolean.valueOf转换，int和Integer类型的属性值（比如"42"）通过Integer.valueOf转换。 value和param不能同时使用，但可以使用其中任意一个。
param	param 是可选的。它指定用哪个请求参数作为Bean属性的值。如果当前请求没有参数，则什么事情也不做，系统不会把null传递给Bean属性的set方法。因此，你可以让Bean自己提供默认属性值，只有当请求参数明确指定了新值时才修改默认属性值。

<jsp:getProperty>动作元素

jsp:getProperty动作提取指定Bean属性的值，转换成字符串，然后输出。语法格式如下：

```
<jsp:useBean id="myName" ... />
...
<jsp:getProperty name="myName" property="someProperty" .../>
```

下表是与getProperty相关联的属性：

属性	描述
name	要检索的Bean属性名称。Bean必须已定义。
property	表示要提取Bean属性的值

实例

以下实例我们使用了Bean:

```
/* 文件: TestBean.java */
package action;

public class TestBean {
   private String message = "No message specified";

   public String getMessage() {
      return(message);
   }
   public void setMessage(String message) {
      this.message = message;
   }
}
```

编译以上实例并生成 TestBean.class 文件，将该文件拷贝至服务器正式存放Java类的目录下，而不是保留给修改后能够自动装载的类的目录(如：C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action目录中，CLASSPATH 变量必须包含该路径。)。例如，对于Java Web Server来说，Bean和所有Bean用到的类都应该放入classes目录，或者封装进jar文件后放入lib目录，但不应该放到servlets 下。 下面是一个很简单的例子，它的功能是装载一个Bean，然后设置/读取它的message属性。

现在让我们在main.jsp文件中调用该Bean:

```
 <html>
<head>
<title>Using JavaBeans in JSP</title>
</head>
<body>
<center>
<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"
                    property="message"
                    value="Hello JSP..." />

<p>Got message....</p>

<jsp:getProperty name="test" property="message" />

</center>
</body>
</html>
```

执行以上文件，输出如下所示：

```
Using JavaBeans in JSP
Got message....
Hello JSP...
```

<jsp:forward> 动作元素

jsp:forward动作把请求转到另外的页面。jsp:forward标记只有一个属性page。语法格式如下所示：

```
<jsp:forward page="Relative URL" />
```

以下是forward相关联的属性：

属性	描述
page	page属性包含的是一个相对URL。page的值既可以直接给出，也可以在请求的时候动态计算，可以是一个JSP页面或者一个 Java Servlet.

实例

以下实例我们使用了两个文件，分别是： date.jps 和 main.jsp。

date.js文件代码如下：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The forward Action Example</title>
</head>
<body>
<center>
<h2>The forward action Example</h2>
<jsp:forward page="date.jsp" />
</center>
</body>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
Today's date: 12-Sep-2010 14:54:22
```

<jsp:plugin>动作元素

jsp:plugin动作用来根据浏览器的类型，插入通过Java插件 运行Java Applet所必需的OBJECT或EMBED元素。

如果需要的插件不存在，它会下载插件，然后执行Java组件。 Java组件可以是一个applet或一个JavaBean。

plugin动作有多个对应HTML元素的属性用于格式化Java 组件。param元素可用于向Applet 或 Bean 传递参数。

以下是使用plugin 动作元素的典型实例:

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"
                           width="60" height="80">
   <jsp:param name="fontcolor" value="red" />
   <jsp:param name="background" value="black" />

   <jsp:fallback>
      Unable to initialize Java Plugin
   </jsp:fallback>

</jsp:plugin>
```

如果你有兴趣可以尝试使用applet来测试jsp:plugin动作元素，<fallback>元素是一个新元素，在组件出现故障的错误是发送给用户错误信息。

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素动态定义XML元素。动态是非常重要的，这就意味着XML元素在编译时是动态生成的而非静态。

以下实例动态定义了XML元素：

```
<%@page language="java" contentType="text/html"%>
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<jsp:element name="xmlElement">
<jsp:attribute name="xmlElementAttr">
   Value for the attribute
</jsp:attribute>
<jsp:body>
   Body for XML element
</jsp:body>
</jsp:element>
</body>
</html>
```

执行时生成HTML代码如下：

```
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<xmlElement xmlElementAttr="Value for the attribute">
   Body for XML element
</xmlElement>
</body>
</html>
```

<jsp:text>动作元素

<jsp:text>动作元素允许在JSP页面和文档中使用写入文本的模板，语法格式如下：

```
<jsp:text>Template data</jsp:text>
```

以上文本模板不能包含其他元素，只能只能包含文本和EL表达式（注：EL表达式将在后续章节中介绍）。请注意，在XML文件中，您不能使用表达式如 ${whatever > 0}，因为>符号是非法的。 你可以使用 ${whatever gt 0}表达式或者嵌入在一个CDATA部分的值。

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

如果你需要在 XHTML 中声明 DOCTYPE,必须使用到<jsp:text>动作元素，实例如下：

```
<jsp:text><![CDATA[<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">]]>
</jsp:text>
<head><title>jsp:text action</title></head>
<body>

<books><book><jsp:text>
    Welcome to JSP Programming
</jsp:text></book></books>

</body>
</html>
```

你可以对以上实例尝试使用<jsp:text>及不使用该动作元素执行结果的区别。

</p　　jsp:setproperty动作有下面四个属性,如下表：<>VM1563:1 Resource interpreted as Image but transferred with MIME type text/html: "http://googleads.g.doubleclick.net/pagead/adview?ai=CYd_viH8rVOrQGoGt8gW1-o…dHQJkV0teBb6uOUXPCSK1MkoVQh3X4TKABpy_58Kajo6JdKAGIQ&sigh=pcHBvVQp9GM&vis=1". ads?client=ca-pub-5751451760833794&format=160x600&output=html&h=600&slotname=4106274865&adk=4689099…:1

JSP 动作元素

与JSP指令元素不同的是，JSP动作元素在请求处理阶段起作用。JSP动作元素是用XML语法写成的。

利用JSP动作可以动态地插入文件、重用JavaBean组件、把用户重定向到另外的页面、为Java插件生成HTML代码。

动作元素只有一种语法，它符合XML标准：

```
<jsp:action_name attribute="value" />
```

动作元素基本上都是预定义的函数，JSP规范定义了一系列的标准动作，它用JSP作为前缀，可用的标准动作元素如下：

语法	描述
jsp:include	在页面被请求的时候引入一个文件。
jsp:useBean	寻找或者实例化一个JavaBean。
jsp:setProperty	设置JavaBean的属性。
jsp:getProperty	输出某个JavaBean的属性。
jsp:forward	把请求转到一个新的页面。
jsp:plugin	根据浏览器类型为Java插件生成OBJECT或EMBED标记。
jsp:element	定义动态XML元素
jsp:attribute	设置动态定义的XML元素属性。
jsp:body	设置动态定义的XML元素内容。
jsp:text	在JSP页面和文档中使用写入文本的模板

常见的属性

所有的动作要素都有两个属性：id属性和scope属性。

- **id**属性：

  id属性是动作元素的唯一标识，可以在JSP页面中引用。动作元素创建的id值可以通过PageContext来调用。

- **scope**属性：

该属性用于识别动作元素的生命周期。 id属性和scope属性有直接关系，scope属性定义了相关联id对象的寿命。 scope属性有四个可能的值： (a) page, (b)request, (c)session, 和 (d) application。

<jsp:include>动作元素

<jsp:include>动作元素用来包含静态和动态的文件。该动作把指定文件插入正在生成的页面。语法格式如下：

```
<jsp:include page="relative URL" flush="true" />
```

前面已经介绍过include指令，它是在JSP文件被转换成Servlet的时候引入文件，而这里的jsp:include动作不同，插入文件的时间是在页面被请求的时候。

以下是include动作相关的属性列表。

属性	描述
page	包含在页面中的相对URL地址。
flush	布尔属性，定义在包含资源前是否刷新缓存区。

实例

以下我们定义了两个文件date.jsp和main.jsp，代码如下所示：

date.jsp文件代码：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:include page="date.jsp" flush="true" />
</center>
</body>
</html>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
The include action Example
Today's date: 12-Sep-2013 14:54:22
```

<jsp:useBean>动作元素

jsp:useBean动作用来装载一个将在JSP页面中使用的JavaBean。

这个功能非常有用，因为它使得我们既可以发挥Java组件重用的优势，同时也避免了损失JSP区别于Servlet的方便性。

jsp:useBean动作最简单的语法为：

```
<jsp:useBean id="name" class="package.class" />
```

在类载入后，我们既可以通过 jsp:setProperty 和 jsp:getProperty 动作来修改和检索bean的属性。

以下是useBean动作相关的属性列表。

属性	描述
class	指定Bean的完整包名。
type	指定将引用该对象变量的类型。
beanName	通过 java.beans.Beans 的 instantiate() 方法指定Bean的名字。

在给出具体实例前，让我们先来看下 jsp:setProperty 和 jsp:getProperty 动作元素：

<jsp:setProperty>动作元素

jsp:setProperty用来设置已经实例化的Bean对象的属性，有两种用法。首先，你可以在jsp:useBean元素的外面（后面）使用jsp:setProperty，如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName" property="someProperty" .../>
```

此时，不管jsp:useBean是找到了一个现有的Bean，还是新创建了一个Bean实例，jsp:setProperty都会执行。第二种用法是把jsp:setProperty放入jsp:useBean元素的内部，如下所示：

```
<jsp:useBean id="myName" ... >
...
   <jsp:setProperty name="myName" property="someProperty" .../>
</jsp:useBean>
```

此时，jsp:setProperty只有在新建Bean实例时才会执行，如果是使用现有实例则不执行jsp:setProperty。

jsp:setproperty动作有下面四个属性,如下表：

属性	描述
name	name属性是必需的。它表示要设置属性的是哪个Bean。
property	property属性是必需的。它表示要设置哪个属性。有一个特殊用法：如果property的值是"*"，表示所有名字和Bean属性名字匹配的请求参数都将被传递给相应的属性set方法。
value	value 属性是可选的。该属性用来指定Bean属性的值。字符串数据会在目标类中通过标准的valueOf方法自动转换成数字、boolean、Boolean、 byte、Byte、char、Character。例如，boolean和Boolean类型的属性值（比如"true"）通过 Boolean.valueOf转换，int和Integer类型的属性值（比如"42"）通过Integer.valueOf转换。 value和param不能同时使用，但可以使用其中任意一个。
param	param 是可选的。它指定用哪个请求参数作为Bean属性的值。如果当前请求没有参数，则什么事情也不做，系统不会把null传递给Bean属性的set方法。因此，你可以让Bean自己提供默认属性值，只有当请求参数明确指定了新值时才修改默认属性值。

<jsp:getProperty>动作元素

jsp:getProperty动作提取指定Bean属性的值，转换成字符串，然后输出。语法格式如下：

```
<jsp:useBean id="myName" ... />
...
<jsp:getProperty name="myName" property="someProperty" .../>
```

下表是与getProperty相关联的属性：

属性	描述
name	要检索的Bean属性名称。Bean必须已定义。
property	表示要提取Bean属性的值

实例

以下实例我们使用了Bean:

```
/* 文件: TestBean.java */
package action;

public class TestBean {
   private String message = "No message specified";

   public String getMessage() {
      return(message);
   }
   public void setMessage(String message) {
      this.message = message;
   }
}
```

编译以上实例并生成 TestBean.class 文件，将该文件拷贝至服务器正式存放Java类的目录下，而不是保留给修改后能够自动装载的类的目录(如：C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action目录中，CLASSPATH 变量必须包含该路径。)。例如，对于Java Web Server来说，Bean和所有Bean用到的类都应该放入classes目录，或者封装进jar文件后放入lib目录，但不应该放到servlets 下。 下面是一个很简单的例子，它的功能是装载一个Bean，然后设置/读取它的message属性。

现在让我们在main.jsp文件中调用该Bean:

```
 <html>
<head>
<title>Using JavaBeans in JSP</title>
</head>
<body>
<center>
<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"
                    property="message"
                    value="Hello JSP..." />

<p>Got message....</p>

<jsp:getProperty name="test" property="message" />

</center>
</body>
</html>
```

执行以上文件，输出如下所示：

```
Using JavaBeans in JSP
Got message....
Hello JSP...
```

<jsp:forward> 动作元素

jsp:forward动作把请求转到另外的页面。jsp:forward标记只有一个属性page。语法格式如下所示：

```
<jsp:forward page="Relative URL" />
```

以下是forward相关联的属性：

属性	描述
page	page属性包含的是一个相对URL。page的值既可以直接给出，也可以在请求的时候动态计算，可以是一个JSP页面或者一个 Java Servlet.

实例

以下实例我们使用了两个文件，分别是： date.jps 和 main.jsp。

date.js文件代码如下：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The forward Action Example</title>
</head>
<body>
<center>
<h2>The forward action Example</h2>
<jsp:forward page="date.jsp" />
</center>
</body>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
Today's date: 12-Sep-2010 14:54:22
```

<jsp:plugin>动作元素

jsp:plugin动作用来根据浏览器的类型，插入通过Java插件 运行Java Applet所必需的OBJECT或EMBED元素。

如果需要的插件不存在，它会下载插件，然后执行Java组件。 Java组件可以是一个applet或一个JavaBean。

plugin动作有多个对应HTML元素的属性用于格式化Java 组件。param元素可用于向Applet 或 Bean 传递参数。

以下是使用plugin 动作元素的典型实例:

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"
                           width="60" height="80">
   <jsp:param name="fontcolor" value="red" />
   <jsp:param name="background" value="black" />

   <jsp:fallback>
      Unable to initialize Java Plugin
   </jsp:fallback>

</jsp:plugin>
```

如果你有兴趣可以尝试使用applet来测试jsp:plugin动作元素，<fallback>元素是一个新元素，在组件出现故障的错误是发送给用户错误信息。

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素动态定义XML元素。动态是非常重要的，这就意味着XML元素在编译时是动态生成的而非静态。

以下实例动态定义了XML元素：

```
<%@page language="java" contentType="text/html"%>
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<jsp:element name="xmlElement">
<jsp:attribute name="xmlElementAttr">
   Value for the attribute
</jsp:attribute>
<jsp:body>
   Body for XML element
</jsp:body>
</jsp:element>
</body>
</html>
```

执行时生成HTML代码如下：

```
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<xmlElement xmlElementAttr="Value for the attribute">
   Body for XML element
</xmlElement>
</body>
</html>
```

<jsp:text>动作元素

<jsp:text>动作元素允许在JSP页面和文档中使用写入文本的模板，语法格式如下：

```
<jsp:text>Template data</jsp:text>
```

以上文本模板不能包含其他元素，只能只能包含文本和EL表达式（注：EL表达式将在后续章节中介绍）。请注意，在XML文件中，您不能使用表达式如 ${whatever > 0}，因为>符号是非法的。 你可以使用 ${whatever gt 0}表达式或者嵌入在一个CDATA部分的值。

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

如果你需要在 XHTML 中声明 DOCTYPE,必须使用到<jsp:text>动作元素，实例如下：

```
<jsp:text><![CDATA[<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">]]>
</jsp:text>
<head><title>jsp:text action</title></head>
<body>

<books><book><jsp:text>
    Welcome to JSP Programming
</jsp:text></book></books>

</body>
</html>
```

你可以对以上实例尝试使用<jsp:text>及不使用该动作元素执行结果的区别。

JSP 隐含对象

JSP隐含对象是JSP容器为每个页面提供的Java对象，开发者可以直接使用它们而不用显式声明。JSP隐含对象也被称为预定义变量。

JSP所支持的九大隐含对象：

对象	描述
request	**HttpServletRequest**类的实例
response	**HttpServletResponse**类的实例
out	**PrintWriter**类的实例，用于把结果输出至网页上
session	**HttpSession**类的实例
application	**ServletContext**类的实例，与应用上下午有关
config	**ServletConfig**类的实例
pageContext	**PageContext**类的实例，提供对JSP页面所有对象以及命名空间的访问
page	类似于Java类中的this关键字
Exception	**Exception**类的对象，代表发生错误的JSP页面中对应的异常对象

request对象

request对象是javax.servlet.http.HttpServletRequest 类的实例。每当客户端请求一个JSP页面时，JSP引擎就会制造一个新的request对象来代表这个请求。

request对象提供了一系列方法来获取HTTP头信息，cookies，HTTP方法等等。

response对象

response对象是javax.servlet.http.HttpServletResponse类的实例。当服务器创建request对象时会同时创建用于响应这个客户端的response对象。

response对象也定义了处理HTTP头模块的接口。通过这个对象，开发者们可以添加新的cookies，时间戳，HTTP状态码等等。

out对象

out对象是 javax.servlet.jsp.JspWriter 类的实例，用来在response对象中写入内容。

最初的JspWriter类对象根据页面是否有缓存来进行不同的实例化操作。可以在page指令中使用buffered='false'属性来轻松关闭缓存。

JspWriter类包含了大部分java.io.PrintWriter类中的方法。不过，JspWriter新增了一些专为处理缓存而设计的方法。还有就是，JspWriter类会抛出IOExceptions异常，而PrintWriter不会。

下表列出了我们将会用来输出boolean，char，int，double，Srtring，object等类型数据的重要方法：

方法	描述
out.print(dataType dt)	输出Type类型的值
out.println(dataType dt)	输出Type类型的值然后换行
out.flush()	刷新输出流

session对象

session对象是 javax.servlet.http.HttpSession 类的实例。和Java Servlets中的session对象有一样的行为。

session对象用来跟踪在各个客户端请求间的会话。

application对象

application对象直接包装了servlet的ServletContext类的对象，是javax.servlet.ServletContext 类的实例。

这个对象在JSP页面的整个生命周期中都代表着这个JSP页面。这个对象在JSP页面初始化时被创建，随着jspDestroy()方法的调用而被移除。

通过向application中添加属性，则所有组成您web应用的JSP文件都能访问到这些属性。

config对象

config对象是 javax.servlet.ServletConfig 类的实例，直接包装了servlet的ServletConfig类的对象。

这个对象允许开发者访问Servlet或者JSP引擎的初始化参数，比如文件路径等。

以下是config对象的使用方法，不是很重要，所以不常用：

```
config.getServletName();
```

它返回包含在<servlet-name>元素中的servlet名字，注意，<servlet-name>元素在 WEB-INF\web.xml 文件中定义。

pageContext 对象

pageContext对象是javax.servlet.jsp.PageContext 类的实例，用来代表整个JSP页面。

这个对象主要用来访问页面信息，同时过滤掉大部分实现细节。

这个对象存储了request对象和response对象的引用。application对象，config对象，session对象，out对象可以通过访问这个对象的属性来导出。

pageContext对象也包含了传给JSP页面的指令信息，包括缓存信息，ErrorPage URL,页面scope等。

PageContext类定义了一些字段，包括PAGE_SCOPE，REQUEST_SCOPE，SESSION_SCOPE， APPLICATION_SCOPE。它也提供了40余种方法，有一半继承自javax.servlet.jsp.JspContext 类。

其中一个重要的方法就是removeArribute()，它可接受一个或两个参数。比如，pageContext.removeArribute("attrName")移除四个scope中相关属性，但是下面这种方法只移除特定scope中的相关属性：

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

page 对象

这个对象就是页面实例的引用。它可以被看做是整个JSP页面的代表。

page 对象就是this对象的同义词。

exception 对象

exception 对象包装了从先前页面中抛出的异常信息。它通常被用来产生对出错条件的适当响应。

JSP 客户端请求

当浏览器请求一个网页时，它会向网络服务器发送一系列不能被直接读取的信息，因为这些信息是作为HTTP信息头的一部分来传送的。您可以查阅HTTP协议来获得更多的信息。

下表列出了浏览器端信息头的一些重要内容，在以后的网络编程中将会经常见到这些信息：

信息	描述
Accept	指定浏览器或其他客户端可以处理的MIME类型。它的值通常为 **image/png** 或 **image/jpeg**
Accept-Charset	指定浏览器要使用的字符集。比如 ISO-8859-1
Accept-Encoding	指定编码类型。它的值通常为 **gzip** 或**compress**
Accept-Language	指定客户端首选语言，servlet会优先返回以当前语言构成的结果集，如果servlet支持这种语言的话。比如 en，en-us，ru等等
Authorization	在访问受密码保护的网页时识别不同的用户
Connection	表明客户端是否可以处理HTTP持久连接。持久连接允许客户端或浏览器在一个请求中获取多个文件。**Keep-Alive** 表示启用持久连接
Content-Length	仅适用于POST请求，表示 POST 数据的字节数
Cookie	返回先前发送给浏览器的cookies至服务器
Host	指出原始URL中的主机名和端口号
If-Modified-Since	表明只有当网页在指定的日期被修改后客户端才需要这个网页。 服务器发送304码给客户端，表示没有更新的资源
If-Unmodified-Since	与If-Modified-Since相反， 只有文档在指定日期后仍未被修改过，操作才会成功
Referer	标志着所引用页面的URL。比如，如果你在页面1，然后点了个链接至页面2，那么页面1的URL就会包含在浏览器请求页面2的信息头中
User-Agent	用来区分不同浏览器或客户端发送的请求，并对不同类型的浏览器返回不同的内容

HttpServletRequest类

request对象是javax.servlet.http.HttpServletRequest类的实例。每当客户端请求一个页面时，JSP引擎就会产生一个新的对象来代表这个请求。

request对象提供了一系列方法来获取HTTP信息头，包括表单数据，cookies，HTTP方法等等。

接下来将会介绍一些在JSP编程中常用的获取HTTP信息头的方法。详细内容请见下表：

方法	描述
Cookie[] getCookies()	返回客户端所有的Cookie的数组
Enumeration getAttributeNames()	返回request对象的所有属性名称的集合
Enumeration getHeaderNames()	返回所有HTTP头的名称集合
Enumeration getParameterNames()	返回请求中所有参数的集合
HttpSession getSession()	返回request对应的session对象，如果没有，则创建一个
HttpSession getSession(boolean create)	返回request对应的session对象，如果没有并且参数create为true，则返回一个新的session对象
Locale getLocale()	返回当前页的Locale对象，可以在response中设置
Object getAttribute(String name)	返回名称为name的属性值，如果不存在则返回null。
ServletInputStream getInputStream()	返回请求的输入流
String getAuthType()	返回认证方案的名称，用来保护servlet，比如 "BASIC" 或者 "SSL" 或 null 如果 JSP没设置保护措施
String getCharacterEncoding()	返回request的字符编码集名称
String getContentType()	返回request主体的MIME类型，若未知则返回null
String getContextPath()	返回request URI中指明的上下文路径
String getHeader(String name)	返回name指定的信息头
String getMethod()	返回此request中的HTTP方法，比如 GET,，POST，或PUT
String getParameter(String name)	返回此request中name指定的参数，若不存在则返回null
String getPathInfo()	返回任何额外的与此request URL相关的路径
String getProtocol()	返回此request所使用的协议名和版本
String getQueryString()	返回此 request URL包含的查询字符串
String getRemoteAddr()	返回客户端的IP地址

String getRemoteHost()	返回客户端的完整名称
String getRemoteUser()	返回客户端通过登录认证的用户，若用户未认证则返回null
String getRequestURI()	返回request的URI
String getRequestedSessionId()	返回request指定的session ID
String getServletPath()	返回所请求的servlet路径
String[] getParameterValues(String name)	返回指定名称的参数的所有值，若不存在则返回null
boolean isSecure()	返回request是否使用了加密通道，比如HTTPS
int getContentLength()	返回request主体所包含的字节数，若未知的返回-1
int getIntHeader(String name)	返回指定名称的request信息头的值
int getServerPort()	返回服务器端口号

HTTP信息头示例

在这个例子中，我们会使用HttpServletRequest类的getHeaderNames()方法来读取HTTP信息头。这个方法以枚举的形式返回当前HTTP请求的头信息。

获取Enumeration对象后，用标准的方式来遍历Enumeration对象，用hasMoreElements()方法来确定什么时候停止，用nextElement()方法来获得每个参数的名字。

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>HTTP Header Request Example</title>
</head>
<body>
<center>
<h2>HTTP Header Request Example</h2>
<table width="100%" border="1" align="center">
<tr bgcolor="#949494">
<th>Header Name</th><th>Header Value(s)</th>
</tr>
<%
   Enumeration headerNames = request.getHeaderNames();
   while(headerNames.hasMoreElements()) {
      String paramName = (String)headerNames.nextElement();
      out.print("<tr><td>" + paramName + "</td>\n");
      String paramValue = request.getHeader(paramName);
      out.println("<td> " + paramValue + "</td></tr>\n");
   }
%>
</table>
</center>
</body>
</html>
```

访问main.jsp，将会得到以下结果：

```
<h1>HTTP Header Request Example</h1>

<table width="100%" border="1" align="center">

<tbody>

<tr><th>Header Name</th><th>Header Value(s)</th></tr>

<tr><td>accept</td><td>*/*</td></tr>

<tr><td>accept-language</td><td>en-us</td></tr>

<tr><td>user-agent</td><td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0

<tr><td>accept-encoding</td><td>gzip, deflate</td></tr>

<tr><td>host</td><td>localhost:8080</td></tr>

<tr><td>connection</td><td>Keep-Alive</td></tr>

<tr><td>cache-control</td><td>no-cache</td></tr>

</tbody>

</table>
```

您可以在上面代码中尝试HttpServletRequest类的其它方法。

JSP 服务器响应

Response响应对象主要将JSP容器处理后的结果传回到客户端。可以通过response变量设置HTTP的状态和向客户端发送数据，如Cookie、HTTP文件头信息等。

一个典型的响应看起来就像下面这样：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包含HTTP版本信息，比如HTTP/1.1，一个状态码，比如200，还有一个非常短的信息对应着状态码，比如OK。

下表摘要出了HTTP1.1响应头中最有用的部分，在网络编程中您将会经常见到它们：

响应头	描述
Allow	指定服务器支持的request方法（GET，POST等等）
Cache-Control	指定响应文档能够被安全缓存的情况。通常取值为 **public**，**private** 或**no-cache** 等等。 Public意味着文档可缓存，Private意味着文档只为单用户服务并且只能使用私有缓存。No-cache 意味着文档不被缓存。
Connection	命令浏览器是否要使用持久的HTTP连接。**close**值** 命令浏览器不使用持久HTTP连接，而keep-alive 意味着使用持久化连接。
Content-Disposition	让浏览器要求用户将响应以给定的名称存储在磁盘中
Content-Encoding	指定传输时页面的编码规则
Content-Language	表述文档所使用的语言，比如en， en-us,，ru等等
Content-Length	表明响应的字节数。只有在浏览器使用持久化 (keep-alive) HTTP 连接时才有用
Content-Type	表明文档使用的MIME类型
Expires	指明啥时候过期并从缓存中移除
Last-Modified	指明文档最后修改时间。客户端可以 缓存文档并且在后续的请求中提供一个 **If-Modified-Since**请求头
Location	在300秒内，包含所有的有一个状态码的响应地址，浏览器会自动重连然后检索新文档
Refresh	指明浏览器每隔多久请求更新一次页面。
Retry-After	与503 (Service Unavailable)一起使用来告诉用户多久后请求将会得到响应
Set-Cookie	指明当前页面对应的cookie

HttpServletResponse类

response对象是javax.servlet.http.HttpServletRequest类的一个实例。就像服务器会创建request对象一样，它也会创建一个客户端响应。

response对象定义了处理创建HTTP信息头的接口。通过使用这个对象，开发者们可以添加新的cookie或时间戳，还有HTTP状态吗等等。

下表列出了用来设置HTTP响应头的方法，这些方法由HttpServletResponse 类提供：

方法	描述
String	

encodeRedirectURL(String url)	对sendRedirect()方法使用的URL进行编码
String encodeURL(String url)	将URL编码，回传包含Session ID的URL
boolean containsHeader(String name)	返回指定的响应头是否存在
boolean isCommitted()	返回响应是否已经提交到客户端
void addCookie(Cookie cookie)	添加指定的cookie至响应中
void addDateHeader(String name, long date)	添加指定名称的响应头和日期值
void addHeader(String name, String value)	添加指定名称的响应头和值
void addIntHeader(String name, int value)	添加指定名称的响应头和int值
void flushBuffer()	将任何缓存中的内容写入客户端
void reset()	清除任何缓存中的任何数据，包括状态码和各种响应头
void resetBuffer()	清除基本的缓存数据，不包括响应头和状态码
void sendError(int sc)	使用指定的状态码向客户端发送一个出错响应，然后清除缓存
void sendError(int sc, String msg)	使用指定的状态码和消息向客户端发送一个出错响应
void sendRedirect(String location)	使用指定的URL向客户端发送一个临时的间接响应
void setBufferSize(int size)	设置响应体的缓存区大小
void setCharacterEncoding(String charset)	指定响应的编码集（MIME字符集），例如UTF-8
void setContentLength(int len)	指定HTTP servlets中响应的内容的长度，此方法用来设置 HTTP Content-Length 信息头
void setContentType(String type)	设置响应的内容的类型，如果响应还未被提交的话
void setDateHeader(String name, long date)	使用指定名称和值设置响应头的名称和内容
void setHeader(String name, String value)	使用指定名称和值设置响应头的名称和内容
void setIntHeader(String name, int value)	使用指定名称和值设置响应头的名称和内容

void setStatus(int sc)	设置响应的状态码

HTTP响应头程序示例

接下来的例子使用setIntHeader()方法和setRefreshHeader()方法来模拟一个数字时钟：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Auto Refresh Header Example</title>
</head>
<body>
<center>
<h2>Auto Refresh Header Example</h2>
<%
   // 设置每隔5秒自动刷新
   response.setIntHeader("Refresh", 5);
   // 获取当前时间
   Calendar calendar = new GregorianCalendar();
   String am_pm;
   int hour = calendar.get(Calendar.HOUR);
   int minute = calendar.get(Calendar.MINUTE);
   int second = calendar.get(Calendar.SECOND);
   if(calendar.get(Calendar.AM_PM) == 0)
      am_pm = "AM";
   else
      am_pm = "PM";
   String CT = hour+":"+ minute +":"+ second +" "+ am_pm;
   out.println("Current Time is: " + CT + "\n");
%>
</center>
</body>
</html>
```

将以上代码保存为main.jsp，然后通过浏览器访问它。它将会每隔5秒显示一下系统当前时间。

运行结果如下：

```
Auto Refresh Header Example
Current Time is: 9:44:50 PM
```

您也可以自己动手修改以上代码，试试使用其他的方法，将能得到更深的体会。

JSP HTTP 状态码

HTTP请求与HTTP响应的格式相近，都有着如下结构：

- 以状态行+CRLF（回车换行）开始
- 零行或多行头模块+CRLF
- 一个空行，比如CRLF
- 可选的消息体比如文件，查询数据，查询输出

举例来说，一个服务器响应头看起来就像下面这样：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包含HTTP版本，一个状态码，和状态码相对应的短消息。

下表列出了可能会从服务器返回的HTTP状态码和与之关联的消息：

状态码	消息	描述
100	Continue	只有一部分请求被服务器接收，但只要没被服务器拒绝，客户端就会延续这个请求
101	Switching Protocols	服务器交换机协议
200	OK	请求被确认
201	Created	请求时完整的，新的资源被创建
202	Accepted	请求被接受，但未处理完
203	Non-authoritative Information	
204	No Content	
205	Reset Content	
206	Partial Content	

300	Multiple Choices	一个超链接表，用户可以选择一个超链接并访问，最大支持5个超链接
301	Moved Permanently	被请求的页面已经移动到了新的URL下
302	Found	被请求的页面暂时性地移动到了新的URL下
303	See Other	被请求的页面可以在一个不同的URL下找到
304	Not Modified	
305	Use Proxy	
306	*Unused*	已经不再使用此状态码，但状态码被保留
307	Temporary Redirect	被请求的页面暂时性地移动到了新的URL下
400	Bad Request	服务器无法识别请求
401	Unauthorized	被请求的页面需要用户名和密码
402	Payment Required	目前还不能使用此状态码
403	Forbidden	禁止访问所请求的页面
404	Not Found	服务器无法找到所请求的页面
405	Method Not Allowed	请求中所指定的方法不被允许
406	Not Acceptable	服务器只能创建一个客户端无法接受的响应
407	Proxy Authentication Required	在请求被服务前必须认证一个代理服务器
408	Request Timeout	请求时间超过了服务器所能等待的时间，连接被断开
409	Conflict	请求有矛盾的地方
410	Gone	被请求的页面不再可用
411	Length Required	"Content-Length"没有被定义，服务器拒绝接受请求
412	Precondition Failed	请求的前提条件被服务器评估为false
413	Request Entity Too Large	因为请求的实体太大，服务器拒绝接受请求
414	Request-url Too Long	服务器拒绝接受请求，因为URL太长。多出现在把"POST"请求转换为"GET"请求时所附带的大量查询信息
415	Unsupported Media Type	服务器拒绝接受请求，因为媒体类型不被支持

417	Expectation Failed	
500	Internal Server Error	请求不完整，服务器遇见了出乎意料的状况
501	Not Implemented	请求不完整，服务器不提供所需要的功能
502	Bad Gateway	请求不完整，服务器从上游服务器接受了一个无效的响应
503	Service Unavailable	请求不完整，服务器暂时重启或关闭
504	Gateway Timeout	网关超时
505	HTTP Version Not Supported	服务器不支持所指定的HTTP版本

设置HTTP状态码的方法

下表列出了HttpServletResponse 类中用来设置状态码的方法：

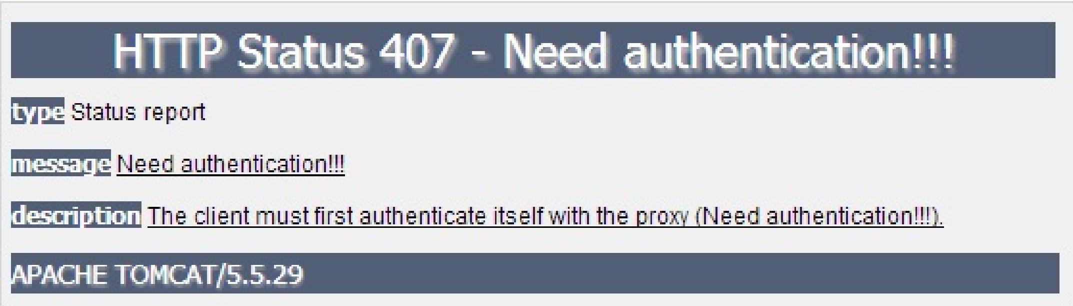
方法	描述
public void setStatus (int statusCode)	此方法可以设置任意的状态码。如果您的响应包含一个特殊的状态码和一个文档，请确保在用PrintWriter返回任何内容前调用setStatus方法
public void sendRedirect(String url)	此方法产生302响应，同时产生一个 *Location* 头告诉URL 一个新的文档
public void sendError(int code, String message)	此方法将一个状态码(通常为 404)和一个短消息，自动插入HTML文档中并发回给客户端

HTTP状态码程序示例

接下来的例子将会发送407错误码给浏览器，然后浏览器将会告诉您"Need authentication!!!"。

```
<html>
<head>
<title>Setting HTTP Status Code</title>
</head>
<body>
<%
   // 设置错误代码，并说明原因
   response.sendError(407, "Need authentication!!!" );
%>
</body>
</html>
```

访问以上JSP页面，将会得到以下结果：



您也可以试试使用其他的状态码，看会不会得到什么意想不到结果。

JSP 表单处理

我们在浏览网页的时候，经常需要向服务器提交信息，并让后台程序处理。浏览器中使用GET 和 POST 方法向服务器提交数据。

GET 方法

GET方法将请求的编码信息添加在网址后面，网址与编码信息通过"?"号分隔。如下所示：

```
http://www.w3cschool.cc/hello?key1=value1&key2=value2
```

GET方法是浏览器默认传递参数的方法，一些敏感信息，如密码等建议不使用GET方法。

用get时，传输数据的大小有限制 （注意不是参数的个数有限制），最大为1024字节。

POST 方法

一些敏感信息，如密码等我们可以同过POST方法传递，post提交数据是隐式的。

POST提交数据是不可见的，GET是通过在url里面传递的（可以看一下你浏览器的地址栏）。

JSP使用getParameter()来获得传递的参数，getInputStream()方法用来处理客户端的二进制数据流的请求。

JSP 读取表单数据

- **getParameter():** 使用 request.getParameter() 方法来获取表单参数的值。
- **getParameterValues():** 获得如checkbox类（名字相同，但值有多个）的数据。 接收数组变量 ，如checkobx类型
- **getParameterNames():**该方法可以取得所有变量的名称，该方法返回一个Emumeration。
- **getInputStream():**调用此方法来读取来自客户端的二进制数据流。

使用URL的 GET 方法实例

以下是一个简单的URL,并使用GET方法来传递URL中的参数：

```
http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI
```

以下是main.jsp文件的JSP程序用于处理客户端提交的表单数据，我们使用getParameter()方法来获取提交的数据：

```
<html>
<head>
<title>Using GET Method to Read Form Data</title>
</head>
<body>
<center>
<h1>Using GET Method to Read Form Data</h1>
<ul>
<li><p><b>First Name:</b>
   <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
   <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

接下来我们通过浏览器访问*http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI* 输出结果如下所示：

```
Using GET Method to Read Form Data
First Name: ZARA

Last Name: ALI
```

使用表单的 GET 方法实例

以下是一个简单的HTML表单，该表单通过GET方法将客户端数据提交 到main.jsp文件中：

```
<html>
<body>
<form action="main.jsp" method="GET">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

将以上HTML代码保存到Hello.htm文件中。 将该文件放置于<Tomcat安装目录>/webapps/ROOT 目录下。 通过访问 *http://localhost:8080/Hello.htm*，输出界面如下所示：

在"First Name" 与 "Last Name"两个表单中填入信息，并点击"Submit"按钮，它将输出结果。

使用表单的 POST 方法实例

接下来让我们使用POST方法来传递表单数据，修改main.jsp与Hello.htm文件代码，如下所示：

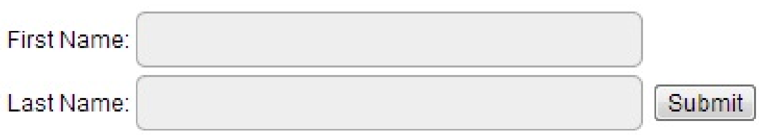
main.jsp文件代码：

```
<html>
<head>
<title>Using GET and POST Method to Read Form Data</title>
</head>
<body>
<center>
<h1>Using GET Method to Read Form Data</h1>
<ul>
<li><p><b>First Name:</b>
   <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
   <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

以下是Hello.htm修改后的代码：

```
<html>
<body>
<form action="main.jsp" method="POST">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

通过浏览器访问 *http://localhost:8080/Hello.htm*，输出如下：



在"First Name" 与 "Last Name"两个表单中填入信息，并点击"Submit"按钮，它将输出结果。

传递 Checkbox 数据到JSP程序

复选框 checkbox 可以传递一个甚至多个数据。

以下是一个简单的HTML代码，并将代码保存在CheckBox.htm文件中：

```
<html>
<body>
<form action="main.jsp" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> Maths
<input type="checkbox" name="physics"  /> Physics
<input type="checkbox" name="chemistry" checked="checked" />
                                                Chemistry
<input type="submit" value="Select Subject" />
</form>
</body>
</html>
```

以上代码在浏览器访问如下所示：

以下为main.jsp文件代码，用于处理复选框数据：

```
<html>
<head>
<title>Reading Checkbox Data</title>
</head>
<body>
<center>
<h1>Reading Checkbox Data</h1>
<ul>
<li><p><b>Maths Flag:</b>
   <%= request.getParameter("maths")%>
</p></li>
<li><p><b>Physics Flag:</b>
   <%= request.getParameter("physics")%>
</p></li>
<li><p><b>Chemistry Flag:</b>
   <%= request.getParameter("chemistry")%>
</p></li>
</ul>
</body>
</html>
```

以上实例输出结果为：

读取所有表单参数

以下我们将使用 HttpServletRequest 的getParameterNames()来读取所有可用的表单参数,该方法可以取得所有变量的名称，该方法返回一个Emumeration。

一旦我们有了一个Enumeration（枚举），我们就可以调用hasMoreElements（）方法来确定何时停止使用和nextElement（）方法来获得每个参数的名称。

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>HTTP Header Request Example</title>
</head>
<body>
<center>
<h2>HTTP Header Request Example</h2>
<table width="100%" border="1" align="center">
<tr bgcolor="#949494">
<th>Param Name</th><th>Param Value(s)</th>
</tr>
<%
   Enumeration paramNames = request.getParameterNames();

   while(paramNames.hasMoreElements()) {
      String paramName = (String)paramNames.nextElement();
      out.print("<tr><td>" + paramName + "</td>\n");
      String paramValue = request.getHeader(paramName);
      out.println("<td> " + paramValue + "</td></tr>\n");
   }
%>
</table>
</center>
</body>
</html>
```

以下是Hello.htm文件的内容:

```
<html>
<body>
<form action="main.jsp" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> Maths
<input type="checkbox" name="physics"  /> Physics
<input type="checkbox" name="chemistry" checked="checked" /> Chem
<input type="submit" value="Select Subject" />
</form>
</body>
</html>
```

现在我们通过浏览器访问 Hello.htm 文件并提交数据，输出结果如下：

Reading All Form Parameters

Param Name	Param Value(s)
maths	on
chemistry	on

你可以尝试使用以上的JSP代码读取其它对象，如文本框，单选按钮或下拉框等等其他形式的数据。

JSP 过滤器

Servlet和JSP中的过滤器都是Java类，它们存在的目的如下：

- 在请求访问后端资源时拦截它
- 管理从服务器返回给客户端的响应

下面列出了多种常用的过滤器类型：

- 认证过滤器
- 数据压缩过滤器
- 加密过滤器
- 触发资源访问事件的过滤器
- 图像转换过滤器
- 登录和验证过滤器
- MIME类型链过滤器
- 令牌过滤器
- 转换XML内容的XSL/T过滤器

过滤器将会被插入进web.xml文件中，然后映射servlet、JSP文件的名字，或URL模式。部署描述文件web.xml可以在 <Tomcat-installation-directory>\conf 目录下找到。

当JSP容器启动网络应用程序时，它会创建每一个过滤器的实例，这些过滤器必须在部署描述文件web.xml中声明，并且按声明的顺序执行。

Servlet过滤器方法

一个过滤器就是一个Java类，它实现了javax.servlet.Filter 接口。javax.servlet.Filter接口定义了三个方法：

方法	描述
public void doFilter (ServletRequest, ServletResponse, FilterChain)	该方法在每次一个请求/响应对因客户端在链的末端请求资源而通过链传递时由容器调用。
public void init(FilterConfig filterConfig)	该方法由 Web 容器调用，指示一个过滤器被放入服务。
public void destroy()	该方法由 Web 容器调用，指示一个过滤器被取出服务。

JSP过滤器示例

这个例子将会打印IP地址和每次访问JSP文件的日期时间。当然，这只是个简单的例子，让您了解一些简单的过滤器用法，但是可以使用这些概念来自行构造更复杂的程序。

```
//  引入Java包
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 实现 Filter 类
public class LogFilter implements Filter  {
   public void  init(FilterConfig config)
                      throws ServletException{
      // 获取初始化参数
      String testParam = config.getInitParameter("test-param");

      //打印初始化参数
      System.out.println("Test Param: " + testParam);
   }
   public void  doFilter(ServletRequest request,
                 ServletResponse response,
                 FilterChain chain)
                 throws java.io.IOException, ServletException {

      // 获取客户端ip地址
      String ipAddress = request.getRemoteAddr();

      // 输出ip地址及当前时间
      System.out.println("IP "+ ipAddress + ", Time "
                                 + new Date().toString());

      // 传递请求道过滤器链
      chain.doFilter(request,response);
   }
   public void destroy( ){
      /* 在Filter实例在服务器上被移除前调用。*/
   }
}
```

编译LogFilter.java文件，然后将编译后的class文件放在<Tomcat安装目录>/webapps/ROOT/WEB-INF/classes目录下。

web.xml文件中的JSP过滤器映射

过滤器被定义，然后映射成一个URL或JSP文件名，与servlet被定义然后映射的方式差不多。在部署描述文件web.xml中，使用<filter>标签来进行过滤器映射：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

上述过滤器将会应用在所有servlet和JSP程序中，因为我们在配置中指定了" /*"。您也可以指定一个servlet或JSP路径，如果您只想要将过滤器应用在少数几个servlet或JSP程序中的话。

现在，像平常一样访问servlet或JSP页面，您就会发现服务器日志中产生了关于此次访问的记录。您也可以使用Log4J记录器来把日志记录在其它文件中。

使用多重过滤器

您的网络应用程序可以定义很多不同的过滤器。现在，您定义了两个过滤器，AuthenFilter和LogFilter，其它的步骤与前面讲的一样，除非要创建一个不同的映射，就像下面这样：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter>
   <filter-name>AuthenFilter</filter-name>
   <filter-class>AuthenFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

过滤器的应用顺序

在web.xml中<filter>元素的映射顺序决定了容器应用这些过滤器的顺序。要反转应用的顺序，您只需要反转web.xml中<filter>元素的定义顺序就行了。

比如，上面的例子会首先应用 LogFilter然后再应用AuthenFilter，但是下面这个例子将会反转应用的顺序：

```
<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

JSP Cookies 处理

Cookies是存储在客户机的文本文件，它们保存了大量轨迹信息。在servlet技术基础上，JSP显然能够提供对HTTP cookies的支持。

通常有三个步骤来识别回头客：

- 服务器脚本发送一系列cookies至浏览器。比如名字，年龄，ID号码等等。
- 浏览器在本地机中存储这些信息，以备不时之需。
- 当下一次浏览器发送任何请求至服务器时，它会同时将这些cookies信息发送给服务器，然后服务器使用这些信息来识别用户或者干些其它事情。

本章节将会传授您如何去设置或重设cookie的方法，还有如何访问它们及如何删除它们。

Cookie剖析

Cookies通常在HTTP信息头中设置（虽然JavaScript能够直接在浏览器中设置cookies）。在JSP中，设置一个cookie需要发送如下的信息头给服务器：

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
                 path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html
```

正如您所见，Set-Cookie信息头包含一个键值对，一个GMT（格林尼治标准）时间，一个路径，一个域名。键值对会被编码为URL。有效期域是个指令，告诉浏览器在什么时候之后就可以清除这个cookie。

如果浏览器被配置成可存储cookies，那么它将会保存这些信息直到过期。如果用户访问的任何页面匹配了cookie中的路径和域名，那么浏览器将会重新将这个cookie发回给服务器。浏览器端的信息头长得就像下面这样：

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

JSP脚本通过request对象中的getCookies()方法来访问这些cookies，这个方法会返回一个Cookie对象的数组。

Servlet Cookies 方法

下表列出了Cookie对象中常用的方法：

方法	描述
public void setDomain(String pattern)	设置cookie的域名，比如w3cschool.cc
public String getDomain()	获取cookie的域名，比如w3cschool.cc
public void setMaxAge(int expiry)	设置cookie有效期，以秒为单位，默认有效期为当前session的存活时间
public int getMaxAge()	获取cookie有效期，以秒为单位，默认为-1 ，表明cookie会活到浏览器关闭为止
public String getName()	返回 cookie的名称，名称创建后将不能被修改
public void setValue(String newValue)	设置 cookie的值
public String getValue()	获取cookie的值
public void setPath(String uri)	设置cookie 的路径，默认为当前页面目录下的所有URL，还有此目录下的所有子目录
public String getPath()	获取cookie 的路径
public void setSecure(boolean flag)	指明cookie是否要加密传输
public void setComment(String purpose)	设置注释描述 cookie的目的。当浏览器将cookie展现给用户时，注释将会变得非常有用
public String getComment()	返回描述cookie目的的注释，若没有则返回null

使用JSP设置Cookies

使用JSP设置cookie包含三个步骤：

(1)创建一个**Cookie**对象： 调用Cookie的构造函数，使用一个cookie名称和值做参数，它们都是字符串。

```
Cookie cookie = new Cookie("key","value");
```

请务必牢记，名称和值中都不能包含空格或者如下的字符：

```
[ ] ( ) = , " / ? @ : ;
```

(2) 设置有效期：调用setMaxAge()函数表明cookie在多长时间（以秒为单位）内有效。下面的操作将有效期设为了24小时。

```
cookie.setMaxAge(60*60*24);
```

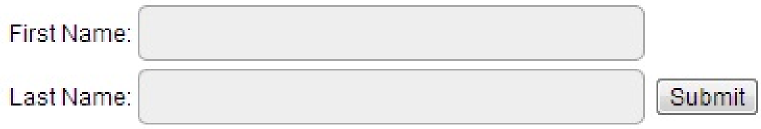
(3) 将**cookie**发送至**HTTP**响应头中：调用response.addCookie()函数来向HTTP响应头中添加cookies。

```
response.addCookie(cookie);
```

实例演示

```
<%
   // 为 first_name 和 last_name设置cookie
   Cookie firstName = new Cookie("first_name",
                request.getParameter("first_name"));
   Cookie lastName = new Cookie("last_name",
               request.getParameter("last_name"));

   // 设置cookie过期时间为24小时。
   firstName.setMaxAge(60*60*24);
   lastName.setMaxAge(60*60*24);

   // 在响应头部添加cookie
   response.addCookie( firstName );
   response.addCookie( lastName );
%>
<html>
<head>
<title>Setting Cookies</title>
</head>
<body>
<center>
<h1>Setting Cookies</h1>
</center>
<ul>
<li><p><b>First Name:</b>
   <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
   <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

将上面两个文件放在<Tomcat安装目录>/webapps/ROOT目录下，然后访问http://localhost:8080/hello.jsp，将会得到如下输出结果：



试着输入First Name和Last Name，然后点击提交按钮，它将会在您的屏幕中显示first name和last name，并且设置first name和last name两个cookie，下一次点击提交按钮时会发给服务器。

使用JSP读取Cookies

想要读取cookies，您就需要调用request.getCookies()方法来获得一个javax.servlet.http.Cookie对象的数组，然后遍历这个数组，使用getName()方法和getValue()方法来获取每一个cookie的名称和值。

让我们来读取上个例子中的cookies。

```
<html>
<head>
<title>Reading Cookies</title>
</head>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
   Cookie cookie = null;
   Cookie[] cookies = null;
   // 获取cookies的数据,是一个数组
   cookies = request.getCookies();
   if( cookies != null ){
      out.println("<h2> Found Cookies Name and Value</h2>");
      for (int i = 0; i < cookies.length; i++){
         cookie = cookies[i];
         out.print("Name : " + cookie.getName( ) + ",  ");
         out.print("Value: " + cookie.getValue( )+" <br/>");
      }
  }else{
      out.println("<h2>No cookies founds</h2>");
  }
%>
</body>
</html>
```

如果您把first name cookie设置成"John"，last name设置成"Player"，访问http://localhost:8080/main.jsp，将会得到如下输出结果：

```
Found Cookies Name and Value
Name : first_name, Value: John
Name : last_name, Value: Player
```

使用JSP删除Cookies

删除cookies非常简单。如果您想要删除一个cookie，按照下面给的步骤来做就行了：

- 获取一个已经存在的cookie然后存储在Cookie对象中。
- 将cookie的有效期设置为0。
- 将这个cookie重新添加进响应头中。

实例演示

下面的程序删除一个名为"first_name"的cookie，当您下次运行main.jsp时，first_name将会为null。

```
<html>
<head>
<title>Reading Cookies</title>
</head>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
   Cookie cookie = null;
   Cookie[] cookies = null;
   // 获取当前域名下的cookies，是一个数组
   cookies = request.getCookies();
   if( cookies != null ){
      out.println("<h2> Found Cookies Name and Value</h2>");
      for (int i = 0; i < cookies.length; i++){
         cookie = cookies[i];
         if((cookie.getName( )).compareTo("first_name") == 0 ){
            cookie.setMaxAge(0);
            response.addCookie(cookie);
            out.print("Deleted cookie: " +
            cookie.getName( ) + "<br/>");
         }
         out.print("Name : " + cookie.getName( ) + ",  ");
         out.print("Value: " + cookie.getValue( )+" <br/>");
      }
  }else{
      out.println(
      "<h2>No cookies founds</h2>");
  }
%>
</body>
</html>
```

访问它，将会得到如下输出结果：

```
Cookies Name and Value
Deleted cookie : first_name
Name : first_name, Value: John
Name : last_name, Value: Player
```

再次访问http://localhost:8080/main.jsp，将会得到如下结果：

```
Found Cookies Name and Value
Name : last_name, Value: Player
```

您也可以手动在浏览器中删除cookies。点击Tools菜单项，然后选择Internet Options，点击Delete Cookies，就能删除所有cookies了。

JSP Session

HTTP是无状态协议，这意味着每次客户端检索网页时，都要单独打开一个服务器连接，因此服务器不会记录下先前客户端请求的任何信息。

有三种方法来维持客户端与服务器的会话：

Cookies

网络服务器可以指定一个唯一的session ID作为cookie来代表每个客户端，用来识别这个客户端接下来的请求。

这可能不是一种有效的方式，因为很多时候浏览器并不一定支持cookie，所以我们不建议使用这种方法来维持会话。

隐藏表单域

一个网络服务器可以发送一个隐藏的HTML表单域和一个唯一的session ID，就像下面这样：

```
<input type="hidden" name="sessionid" value="12345">
```

这个条目意味着，当表单被提交时，指定的名称和值将会自动包含在GET或POST数据中。每当浏览器发送一个请求，session_id的值就可以用来保存不同浏览器的轨迹。

这种方式可能是一种有效的方式，但点击<A HREF>标签中的超链接时不会产生表单提交事件，因此隐藏表单域也不支持通用会话跟踪。

重写URL

您可以在每个URL后面添加一些额外的数据来区分会话，服务器能够根据这些数据来关联session标识符。

举例来说，http://w3cschool.cc/file.htm;sessionid=12345， session标识符为sessionid=12345，服务器可以用这个数据来识别客户端。

相比而言，重写URL是更好的方式来，就算浏览器不支持cookies也能工作，但缺点是您必须为每个URL动态指定session ID，就算这是个简单的HTML页面。

session对象

除了以上几种方法外，JSP利用servlet提供的HttpSession接口来识别一个用户，存储这个用户的所有访问信息。

默认情况下，JSP允许会话跟踪，一个新的HttpSession对象将会自动地为新的客户端实例化。禁止会话跟踪需要显式地关掉它，通过将page指令中session属性值设为false来实现，就像下面这样：

```
<%@ page session="false" %>
```

JSP引擎将隐含的session对象暴露给开发者。由于提供了session对象，开发者就可以方便地存储或检索数据。

下表列出了session对象的一些重要方法：

方法	描述
public Object getAttribute(String name)	返回session对象中与指定名称绑定的对象，如果不存在则返回null
public Enumeration getAttributeNames()	返回session对象中所有的对象名称
public long getCreationTime()	返回session对象被创建的时间， 以毫秒为单位，从1970年1月1号凌晨开始算起
public String getId()	返回session对象的ID
public long getLastAccessedTime()	返回客户端最后访问的时间，以毫秒为单位，从1970年1月1号凌晨开始算起
public int getMaxInactiveInterval()	返回最大时间间隔，以秒为单位，servlet 容器将会在这段时间内保持会话打开
public void invalidate()	将session无效化，解绑任何与该session绑定的对象
public boolean isNew(	返回是否为一个新的客户端，或者客户端是否拒绝加入session
public void removeAttribute(String name)	移除session中指定名称的对象
public void setAttribute(String name, Object value)	使用指定的名称和值来产生一个对象并绑定到session中
public void setMaxInactiveInterval(int interval)	用来指定时间，以秒为单位，servlet容器将会在这段时间内保持会话有效

JSP Session应用

这个例子描述了如何使用HttpSession对象来获取创建时间和最后一次访问时间。我们将会为request对象关联一个新的session对象，如果这个对象尚未存在的话。

```
<%@ page import="java.io.*,java.util.*" %>
<%
   // 获取session创建时间
   Date createTime = new Date(session.getCreationTime());
   // 获取最后访问页面的时间
   Date lastAccessTime = new Date(session.getLastAccessedTime());

   String title = "Welcome Back to my website";
   Integer visitCount = new Integer(0);
   String visitCountKey = new String("visitCount");
   String userIDKey = new String("userID");
   String userID = new String("ABCD");

   // 检测网页是否由新的访问用户
   if (session.isNew()){
      title = "Welcome to my website";
      session.setAttribute(userIDKey, userID);
      session.setAttribute(visitCountKey,  visitCount);
   }
   visitCount = (Integer)session.getAttribute(visitCountKey);
   visitCount = visitCount + 1;
   userID = (String)session.getAttribute(userIDKey);
   session.setAttribute(visitCountKey,  visitCount);
%>
<html>
<head>
<title>Session Tracking</title>
</head>
<body>
<center>
<h1>Session Tracking</h1>
</center>
<table border="1" align="center">
<tr bgcolor="#949494">
   <th>Session info</th>
   <th>Value</th>
</tr>
<tr>
   <td>id</td>
   <td><% out.print( session.getId()); %></td>
</tr>
<tr>
   <td>Creation Time</td>
   <td><% out.print(createTime); %></td>
</tr>
<tr>
   <td>Time of Last Access</td>
   <td><% out.print(lastAccessTime); %></td>
</tr>
<tr>
   <td>User ID</td>
   <td><% out.print(userID); %></td>
</tr>
<tr>
   <td>Number of visits</td>
   <td><% out.print(visitCount); %></td>
</tr>
</table>
</body>
</html>
```

试着访问http://localhost:8080/main.jsp，第一次运行时将会得到如下结果：

Welcome to my website

Session Infomation

Session info	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	0

再次访问，将会得到如下结果：

Welcome Back to my website

Session Infomation

info type	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	1

删除Session数据

当处理完一个用户的会话数据后，您可以有如下选择：

- 移除一个特定的属性：

  调用public void removeAttribute(String name) 方法来移除指定的属性。

- 删除整个会话：

  调用public void invalidate() 方法来使整个session无效。

- 设置会话有效期：

  调用 public void setMaxInactiveInterval(int interval) 方法来设置session超时。

- 登出用户：

  支持servlet2.4版本的服务器，可以调用 logout()方法来登出用户，并且使所有相关的session无效。

- 配置**web.xml**文件：

如果使用的是Tomcat，可以向下面这样配置web.xml文件：

```
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
```

超时以分钟为单位，Tomcat中的默认的超时时间是30分钟。

Servlet中的getMaxInactiveInterval( ) 方法以秒为单位返回超时时间。如果在web.xml中配置的是15分钟，则getMaxInactiveInterval( ) 方法将会返回900。

JSP 文件上传

JSP可以通过HTML的form表单上传文件到服务器。 文件类型可以是文本文件、二进制文件、图像文件等其他任何文档。

创建文件上传表单

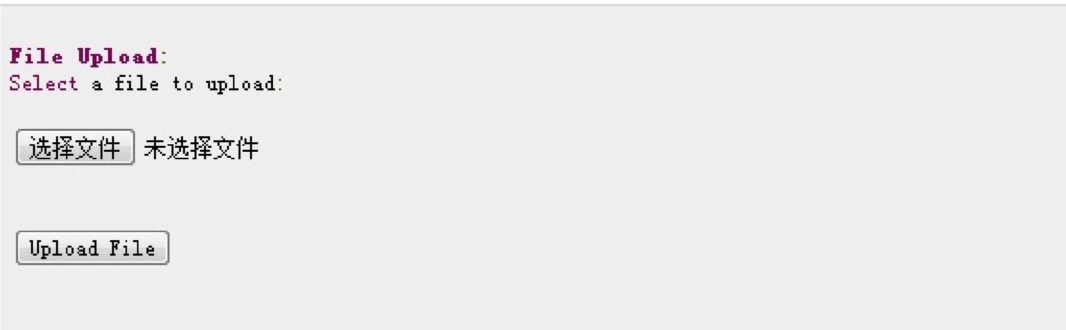
接下来我们使用HTML标签来创建文件上传表单，以下为要注意的点：

- form表单 **method** 属性必须设置为 **POST** 方法， 不能使用 GET 方法。
- form表单 **enctype** 属性需要设置为 **multipart/form-data**。
- form表单 **action** 属性需要设置为提交到后台处理文件上传的jsp文件地址。例如 **uploadFile.jsp** 程序文件用来处理上传的文件。
- 上传文件元素需要使用 <input .../> 标签，属性设置为 type="file"。如果需要上传多个文件，可以在 <input .../>标签中设置不同的名称。

以下是一个上传文件的表单，实例如下：

```
<html>
<head>
<title>File Uploading Form</title>
</head>
<body>
<h3>File Upload:</h3>
Select a file to upload: <br />
<form action="UploadServlet" method="post"
                        enctype="multipart/form-data">
<input type="file" name="file" size="50" />
<br />
<input type="submit" value="Upload File" />
</form>
</body>
</html>
```

在你本地浏览器访问该文件，显示界面如下所示，在你点击"Upload File"会弹出一个窗口让你选择要上传的文件：



后台JSP处理脚本

首先我们先定义文件上传后存储在服务上的位置，你可以将路径写在你的程序当中，或者我们可以在web.xml配置文件中通过设置 context-param 元素来设置文件存储的目录，如下所示：

```
<web-app>
....
<context-param>
    <description>Location to store uploaded file</description>
    <param-name>file-upload</param-name>
    <param-value>
         c:\apache-tomcat-5.5.29\webapps\data\
     </param-value>
</context-param>
....
</web-app>
```

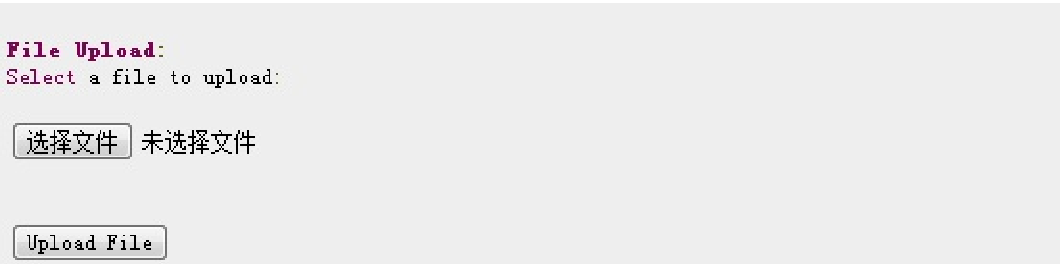
以下脚本文件UploadFile.jsp可以处理多个上传的文件，在使用该脚本前，我们需要注意以下几点：

- 以下实例依赖 FileUpload, 所以你需要在你的classpath中引入最新的 **commons-fileupload.x.x.jar** 包文件。 下载地址为：http://commons.apache.org/fileupload/。
- FileUpload 依赖 Commons IO, 所以你需要在你的classpath中引入最新的 **commons-io-x.x.jar** 。 下载地址为： http://commons.apache.org/io/。
- 在测试以下实例时，你需要上传确认上传的文件大小小于 *maxFileSize* 变量设置的大小，否则文件无法上传成功。
- 确保你已经创建了目录 c:\temp 和 c:\apache-tomcat-5.5.29\webapps\data 。

```
<%@ page import="java.io.*,java.util.*, javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>
<%@ page import="org.apache.commons.fileupload.*" %>
<%@ page import="org.apache.commons.fileupload.disk.*" %>
<%@ page import="org.apache.commons.fileupload.servlet.*" %>
<%@ page import="org.apache.commons.io.output.*" %>

<%
   File file ;
   int maxFileSize = 5000 * 1024;
   int maxMemSize = 5000 * 1024;
   ServletContext context = pageContext.getServletContext();
   String filePath = context.getInitParameter("file-upload");

   // 验证上传内容了类型
   String contentType = request.getContentType();
   if ((contentType.indexOf("multipart/form-data") >= 0)) {

      DiskFileItemFactory factory = new DiskFileItemFactory();
      // 设置内存中存储文件的最大值
      factory.setSizeThreshold(maxMemSize);
      // 本地存储的数据大于 maxMemSize.
      factory.setRepository(new File("c:\\temp"));

      // 创建一个新的文件上传处理程序
      ServletFileUpload upload = new ServletFileUpload(factory);
      // 设置最大上传的文件大小
      upload.setSizeMax( maxFileSize );
      try{
```

```
            // 解析获取的文件
            List fileItems = upload.parseRequest(request);

            // 处理上传的文件
            Iterator i = fileItems.iterator();

            out.println("<html>");
            out.println("<head>");
            out.println("<title>JSP File upload</title>");
            out.println("</head>");
            out.println("<body>");
            while ( i.hasNext () )
            {
               FileItem fi = (FileItem)i.next();
               if ( !fi.isFormField () )
               {
               // 获取上传文件的参数
               String fieldName = fi.getFieldName();
               String fileName = fi.getName();
               boolean isInMemory = fi.isInMemory();
               long sizeInBytes = fi.getSize();
               // 写入文件
               if( fileName.lastIndexOf("\\") >= 0 ){
               file = new File( filePath ,
               fileName.substring( fileName.lastIndexOf("\\"))) ;
               }else{
               file = new File( filePath ,
               fileName.substring(fileName.lastIndexOf("\\")+1)) ;
               }
               fi.write( file ) ;
               out.println("Uploaded Filename: " + filePath +
               fileName + "<br>");
               }
            }
            out.println("</body>");
            out.println("</html>");
         }catch(Exception ex) {
            System.out.println(ex);
         }
      }else{
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet upload</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<p>No file uploaded</p>");
         out.println("</body>");
         out.println("</html>");
      }
%>
```

接下来让我们通过浏览器访问 *http://localhost:8080/UploadFile.htm*，界面如下所示，并上传文件：



如果你的JSP脚本运行正常，文件将被上传至 c:\apache-tomcat-5.5.29\webapps\data\ ，你可以打开文件夹看看是否上传成功。

JSP 日期处理

使用JSP最重要的优势之一，就是可以使用所有Java API。本章将会详细地讲述Java中的Date类，它在java.util包下，封装了当前日期和时间。

Date类有两个构造函数。第一个构造函数使用当前日期和时间来初始化对象。

```
Date( )
```

第二个构造函数接受一个参数，这个参数表示从1970年1月1日凌晨至所要表示时间的毫秒数。

```
Date(long millisec)
```

获取Date对象后，您就能够使用下表列出的所有方法：

方法	描述
boolean after(Date date)	如果比给定的日期晚，则返回true，否则返回false
boolean before(Date date)	如果比给定的日期早，则返回true，否则返回false
Object clone( )	获取当前对象的一个副本
int compareTo(Date date)	如果与给定日期相等，则返回0，如果比给定日期早，则返回一个负数，如果比给定日期晚，则返回一个正数
int compareTo(Object obj)	与 compareTo(Date) 方法相同，如果 obj 不是Date类或其子类的对象，抛出ClassCastException异常
boolean equals(Object date)	如果与给定日期相同，则返回true，否则返回false
long getTime( )	返回从1970年1月1日凌晨至此对象所表示时间的毫秒数
int hashCode( )	返回此对象的哈希码
void setTime(long time)	使用给定参数设置时间和日期，参数time表示从1970年1月1日凌晨至time所经过的毫秒数
String toString( )	将此对象转换为字符串并返回这个字符串

获取当前日期和时间

使用JSP编程可以很容易的获取当前日期和时间，只要使用Date对象的toString()方法就行了，就像下面这样：

```
<%@ page import="java.io.*,java.util.*, javax.servlet.*" %>
<html>
<head>
<title>Display Current Date & Time</title>
</head>
<body>
<center>
<h1>Display Current Date & Time</h1>
</center>
<%
   Date date = new Date();
   out.print( "<h2 align=\"center\">" +date.toString()+"</h2>");
%>
</body>
</html>
```

将上面的代码保存在CurrentDate.jsp文件中，然后访问http://localhost:8080/CurrentDate.jsp，运行结果如下：

```
Display Current Date & Time
Mon Jun 21 21:46:49 GMT+04:00 2013
```

刷新http://localhost:8080/CurrentDate.jsp，就可以发现每次刷新所得到的秒数都不相同。

日期比较

就像我在开头所提到的，您可以在JSP脚本中使用任何Java方法。如果您想要比较两个日期，

可以参照下面的方法来做：

- 使用getTime()方法得到毫秒数，然后比较毫秒数就行了。
- 使用before()，after()，equals()方法。比如，new Date(99,2,12).before(new Date(99,2,18))返回true。
- 使用compareTo()方法，这个方法在Comparable接口中定义，在Date中实现。

使用SimpleDateFormat格式化日期

SimpleDateFormat使用一种地区敏感的方式来格式化和解析日期，它允许您使用自定义的模式来格式化日期和时间。

对CurrentDate.jsp稍作修改，得到如下修改后的代码：

```
<%@ page import="java.io.*,java.util.*" %>
<%@ page import="javax.servlet.*,java.text.*" %>
<html>
<head>
<title>Display Current Date & Time</title>
</head>
<body>
<center>
<h1>Display Current Date & Time</h1>
</center>
<%
   Date dNow = new Date( );
   SimpleDateFormat ft =
   new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
   out.print( "<h2 align=\"center\">" + ft.format(dNow) + "</h2>");
%>
</body>
</html>
```

再次编译CurrentDate.jsp，然后访问http://localhost:8080/CurrentDate.jsp，就可以得到如下结果：

```
Display Current Date & Time
Mon 2013.06.21 at 10:06:44 PM GMT+04:00
```

SimpleDateFormat格式码

要指定模式字符串，需要使用下表列出的格式码：

字符	描述	示例
G	时代标识符	AD
y	4位数年份	2001
M	月	July or 07
d	日	10
h	12小时制， A.M./P.M. (1~12)	12
H	24小时制	22
m	分钟	30
s	秒	55
S	毫秒	234
E	星期	Tuesday
D	一年中的某天	360
F	一个月中某星期的某天	2 (second Wed. in July)
w	一年中的某星期	40
W	一个月中的某星期	1
a	A.M./P.M. 标记	PM
k	一天中的某个小时 (1~24)	24
K	一天中的某个小时，A.M./P.M. (0~11)	10
z	时区	Eastern Standard Time
'	文本分隔	Delimiter
"	单引号	`

更多关于Date类的详细信息请查阅Java API文档。

JSP 页面重定向

当需要将文档移动到一个新的位置时，就需要使用JSP重定向了。

最简单的重定向方式就是使用response对象的sendRedirect()方法。这个方法的签名如下：

```
public void response.sendRedirect(String location)
throws IOException
```

这个方法将状态码和新的页面位置作为响应发回给浏览器。您也可以使用setStatus()和setHeader()方法来得到同样的效果：

```
....
String site = "http://www.w3cschool.cc" ;
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
....
```

实例演示

这个例子表明了JSP如何进行页面重定向：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Page Redirection</title>
</head>
<body>
<center>
<h1>Page Redirection</h1>
</center>
<%
   // 重定向到新地址
   String site = new String("http://www.w3cschool.cc");
   response.setStatus(response.SC_MOVED_TEMPORARILY);
   response.setHeader("Location", site);
%>
</body>
</html>
```

将以上代码保存在PageRedirecting.jsp文件中，然后访问http://localhost:8080/PageRedirect.jsp，它将会把您带至http://www.w3cschool.cc/。

JSP 点击量统计

有时候我们需要知道某个页面被访问的次数，这时我们就需要在页面上添加页面统计器，页面访问的统计一般在用户第一次载入时累加该页面的访问数上。

要实现一个计数器，您可以利用应用程序隐式对象和相关方法getAttribute()和setAttribute()来实现。

这个对象表示JSP页面的整个生命周期中。当JSP页面初始化时创建此对象，当JSP页面调用jspDestroy()时删除该对象。

以下是在应用中创建变量的语法：

```
application.setAttribute(String Key, Object Value);
```

您可以使用上述方法来设置一个计数器变量及更新该变量的值。读取该变量的方法如下：

```
application.getAttribute(String Key);
```

在页面每次被访问时，你可以读取计数器的当前值，并递增1，然后重新设置，在下一个用户访问时就将新的值显示在页面上。

实例演示

该实例将介绍如何使用JSP来计算特定页面访问的总人数。如果你要计算你网站使用页面的总点击量，那么你就必须将该代码放在所有的JSP页面上。

```
<%@ page import="java.io.*,java.util.*" %>

<html>
<head>
<title>Applcation object in JSP</title>
</head>
<body>
<%
    Integer hitsCount =
      (Integer)application.getAttribute("hitCounter");
    if( hitsCount ==null || hitsCount == 0 ){
       /* 第一次访问 */
       out.println("Welcome to my website!");
       hitsCount = 1;
    }else{
       /* 返回访问值 */
       out.println("Welcome back to my website!");
       hitsCount += 1;
    }
    application.setAttribute("hitCounter", hitsCount);
%>
<center>
<p>Total number of visits: <%= hitsCount%></p>
</center>
</body>
</html>
```

现在我们将上面的代码放置于main.jsp文件上，并访问*http://localhost:8080/main.jsp*文件。你会看到页面会生成个计数器，在我们每次刷新页面时，计数器都会发生变化（每次刷新增加1）。 你也可以通过不同的浏览器访问，计数器会在每次访问后增加1。如下所示：

```
Welcome back to my website!

Total number of visits: 12
```

复位计数器

使用以上方法，在web服务器重启后，计数器会被复位为0，即前面保留的数据都会消失，你可以使用一下几种方式解决该问题:

- 在数据库中定义一个用于统计网页访问量的数据表count，字段为hitcount，hitcount默认值为0，将统计数据写入到数据表中。
- 在每次访问时我们读取表中hitcount字段。
- 每次访问时让hitcount自增1。
- 在页面上显示新的 hitcount 值作为页面的访问量。
- 如果你需要统计每个页面的访问量，你可以使用以上逻辑将代码添加到所有页面上。

JSP 自动刷新

想象一下，如果要直播比赛的比分，或股票市场的实时状态，或当前的外汇配给，该怎么实现呢？显然，要实现这种实时功能，您就不得不规律性地刷新页面。

JSP提供了一种机制来使这种工作变得简单，它能够定时地自动刷新页面。

刷新一个页面最简单的方式就是使用response对象的setIntHeader()方法。这个方法的签名如下：

```
public void setIntHeader(String header, int headerValue)
```

这个方法通知浏览器在给定的时间后刷新，时间以秒为单位。

页面自动刷新程序示例

这个例子使用了setIntHeader()方法来设置刷新头，模拟一个数字时钟：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Auto Refresh Header Example</title>
</head>
<body>
<center>
<h2>Auto Refresh Header Example</h2>
<%
   // Set refresh, autoload time as 5 seconds
   response.setIntHeader("Refresh", 5);
   // Get current time
   Calendar calendar = new GregorianCalendar();
   String am_pm;
   int hour = calendar.get(Calendar.HOUR);
   int minute = calendar.get(Calendar.MINUTE);
   int second = calendar.get(Calendar.SECOND);
   if(calendar.get(Calendar.AM_PM) == 0)
      am_pm = "AM";
   else
      am_pm = "PM";
   String CT = hour+":"+ minute +":"+ second +" "+ am_pm;
   out.println("Crrent Time: " + CT + "\n");
%>
</center>
</body>
</html>
```

把以上代码保存在main.jsp文件中，访问它。它会每隔5秒钟刷新一次页面并获取系统当前时间。运行结果如下：

```
Auto Refresh Header Example
Current Time is: 9:44:50 PM
```

您也可以自己动手写个更复杂点的程序。

JSP 发送邮件

虽然使用JSP实现邮件发送功能很简单，但是需要有JavaMail API，并且需要安装JavaBean Activation Framework。

- 在这里下载最新版本的 JavaMail。
- 在这里下载最新版本的 JavaBeans Activation Framework(JAF)。

下载并解压这些文件，在根目录下，您将会看到一系列jar包。将mail.jar包和activation.jar包加入CLASSPATH变量中。

发送一封简单的邮件

这个例子展示了如何从您的机器发送一封简单的邮件。它假定localhost已经连接至网络并且有能力发送一封邮件。与此同时，请再一次确认mail.jar包和activation.jar包已经添加进CLASSPATH变量中。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
   String result;
   // 收件人的电子邮件
   String to = "abcd@gmail.com";

   // 发件人的电子邮件
   String from = "mcmohd@gmail.com";

   // 假设你是从本地主机发送电子邮件
   String host = "localhost";

   // 获取系统属性对象
   Properties properties = System.getProperties();

   // 设置邮件服务器
   properties.setProperty("mail.smtp.host", host);

   // 获取默认的Session对象。
   Session mailSession = Session.getDefaultInstance(properties);

   try{
      // 创建一个默认的MimeMessage对象。
      MimeMessage message = new MimeMessage(mailSession);
      // 设置 From: 头部的header字段
      message.setFrom(new InternetAddress(from));
      // 设置 To: 头部的header字段
      message.addRecipient(Message.RecipientType.TO,
                               new InternetAddress(to));
      // 设置 Subject: header字段
      message.setSubject("This is the Subject Line!");
      // 现在设置的实际消息
      message.setText("This is actual message");
      // 发送消息
      Transport.send(message);
      result = "Sent message successfully....";
   }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
%>
<html>
<head>
<title>Send Email using JSP</title>
</head>
<body>
<center>
<h1>Send Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

现在访问 http://localhost:8080/SendEmail.jsp ，它将会发送一封邮件给abcd@gmail.com 并显示如下结果：

```
Send Email using JSP
Result: Sent message successfully....
```

如果想要把邮件发送给多人，下面列出的方法可以用来指明多个邮箱地址：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

参数的描述如下：

- type：这个值将会被设置成TO，CC,或BCC。CC代表副本，BCC代表黑色副本，例子程序中使用的是TO。
- addresses：这是一个邮箱地址的数组，当指定邮箱地址时需要使用InternetAddress()方法。

发送一封HTML邮件

这个例子发送一封简单的HTML邮件。它假定您的localhost已经连接至网络并且有能力发送邮件。与此同时，请再一次确认mail.jar包和activation.jar包已经添加进CLASSPATH变量中。

这个例子和前一个例子非常相似，不过在这个例子中我们使用了setContent()方法，将"text/html"做为第二个参数传给它，用来表明消息中包含了HTML内容。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
   String result;
   // 收件人的电子邮件
   String to = "abcd@gmail.com";

   // 发件人的电子邮件
   String from = "mcmohd@gmail.com";

   // 假设你是从本地主机发送电子邮件
   String host = "localhost";

   // 获取系统属性对象
   Properties properties = System.getProperties();

   // 设置邮件服务器
   properties.setProperty("mail.smtp.host", host);

   // 获取默认的Session对象。
   Session mailSession = Session.getDefaultInstance(properties);

   try{
      // 创建一个默认的MimeMessage对象。
      MimeMessage message = new MimeMessage(mailSession);
      // 设置 From: 头部的header字段
      message.setFrom(new InternetAddress(from));
      // 设置 To: 头部的header字段
      message.addRecipient(Message.RecipientType.TO,
                               new InternetAddress(to));
      // 设置 Subject: header字段
      message.setSubject("This is the Subject Line!");

      // 设置 HTML消息
      message.setContent("<h1>This is actual message</h1>",
                            "text/html" );
      // 发送消息
      Transport.send(message);
      result = "Sent message successfully....";
   }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
%>
<html>
<head>
<title>Send HTML Email using JSP</title>
</head>
<body>
<center>
<h1>Send Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

现在你可以尝试使用以上JSP文件来发送HTML消息的电子邮件。

在邮件中包含附件

这个例子告诉我们如何发送一封包含附件的邮件。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
   String result;
   // 收件人的电子邮件
   String to = "abcd@gmail.com";

   // 发件人的电子邮件
   String from = "mcmohd@gmail.com";

   // 假设你是从本地主机发送电子邮件
   String host = "localhost";

   // 获取系统属性对象
   Properties properties = System.getProperties();

   // 设置邮件服务器
   properties.setProperty("mail.smtp.host", host);

   // 获取默认的Session对象。
   Session mailSession = Session.getDefaultInstance(properties);

   try{
      // 创建一个默认的MimeMessage对象。
      MimeMessage message = new MimeMessage(mailSession);

      // 设置 From: 头部的header字段
      message.setFrom(new InternetAddress(from));

      // 设置 To: 头部的header字段
      message.addRecipient(Message.RecipientType.TO,
                               new InternetAddress(to));

      // 设置 Subject: header字段
      message.setSubject("This is the Subject Line!");

      // 创建消息部分
      BodyPart messageBodyPart = new MimeBodyPart();

      // 填充消息
      messageBodyPart.setText("This is message body");

      // 创建多媒体消息
      Multipart multipart = new MimeMultipart();

      // 设置文本消息部分
      multipart.addBodyPart(messageBodyPart);

      // 附件部分
      messageBodyPart = new MimeBodyPart();
      String filename = "file.txt";
      DataSource source = new FileDataSource(filename);
      messageBodyPart.setDataHandler(new DataHandler(source));
      messageBodyPart.setFileName(filename);
      multipart.addBodyPart(messageBodyPart);

      // 发送完整消息
      message.setContent(multipart );

      // 发送消息
      Transport.send(message);
      String title = "Send Email";
      result = "Sent message successfully....";
   }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
```

```
%>
<html>
<head>
<title>Send Attachement Email using JSP</title>
</head>
<body>
<center>
<h1>Send Attachement Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

用户认证部分

如果邮件服务器需要用户名和密码来进行用户认证的话，可以像下面这样来设置：

```
 props.setProperty("mail.user", "myuser");
 props.setProperty("mail.password", "mypwd");
```

使用表单发送邮件

使用HTML表单接收一封邮件，并通过request对象获取所有邮件信息：

```
String to = request.getParameter("to");
String from = request.getParameter("from");
String subject = request.getParameter("subject");
String messageText = request.getParameter("body");
```

获取以上信息后，您就可以使用前面提到的例子来发送邮件了。

JSP 高级教程

JSP 标准标签库（JSTL）

JSP标准标签库（JSTL）是一个JSP标签集合，它封装了JSP应用的通用核心功能。

JSTL支持通用的、结构化的任务，比如迭代，条件判断，XML文档操作，国际化标签，SQL标签。 除了这些，它还提供了一个框架来使用集成JSTL的自定义标签。

根据JSTL标签所提供的功能，可以将其分为5个类别。

- 核心标签
- 格式化标签
- **SQL** 标签
- **XML** 标签
- **JSTL** 函数

JSTL 库安装

Apache Tomcat安装JSTL 库步骤如下：

- 从Apache的标准标签库中下载的二进包(jakarta-taglibs-standard-current.zip)。下载地址：http://archive.apache.org/dist/jakarta/taglibs/standard/binaries/
- 下载jakarta-taglibs-standard-1.1.1.zip 包并解压，将jakarta-taglibs-standard-1.1.1/lib/下的两个jar文件：standard.jar和jstl.jar文件拷贝到/WEB-INF/lib/下。

使用任何库，你必须在每个JSP文件中的头部包含<taglib>标签。

核心标签

核心标签是最常用的JSTL标签。引用核心标签库的语法如下：

```
<%@ taglib prefix="c"
           uri="http://java.sun.com/jsp/jstl/core" %>
```

标签	描述
<c:out>	用于在JSP中显示数据，就像<%= ... >
<c:set>	用于保存数据
<c:remove>	用于删除数据
<c:catch>	用来处理产生错误的异常状况，并且将错误信息储存起来
<c:if>	与我们在一般程序中用的if一样
<c:choose>	本身只当做<c:when>和<c:otherwise>的父标签
<c:when>	<c:choose>的子标签，用来判断条件是否成立
<c:otherwise>	<c:choose>的子标签，接在<c:when>标签后，当<c:when>标签判断为false时被执行
<c:import>	检索一个绝对或相对 URL，然后将其内容暴露给页面
<c:forEach>	基础迭代标签，接受多种集合类型
<c:forTokens>	根据指定的分隔符来分隔内容并迭代输出
<c:param>	用来给包含或重定向的页面传递参数
<c:redirect>	重定向至一个新的URL.
<c:url>	使用可选的查询参数来创造一个URL

格式化标签

JSTL格式化标签用来格式化并输出文本、日期、时间、数字。引用格式化标签库的语法如下：

```
<%@ taglib prefix="fmt"
           uri="http://java.sun.com/jsp/jstl/fmt" %>
```

标签	描述
<fmt:formatNumber>	使用指定的格式或精度格式化数字
<fmt:parseNumber>	解析一个代表着数字，货币或百分比的字符串
<fmt:formatDate>	使用指定的风格或模式格式化日期和时间
<fmt:parseDate>	解析一个代表着日期或时间的字符串
<fmt:bundle>	绑定资源
<fmt:setLocale>	指定地区
<fmt:setBundle>	绑定资源
<fmt:timeZone>	指定时区
<fmt:setTimeZone>	指定时区
<fmt:message>	显示资源配置文件信息
<fmt:requestEncoding>	设置request的字符编码

SQL标签

JSTL SQL标签库提供了与关系型数据库（Oracle，MySQL，SQL Server等等）进行交互的标签。引用SQL标签库的语法如下：

```
<%@ taglib prefix="sql"
           uri="http://java.sun.com/jsp/jstl/sql" %>
```

标签	描述
<sql:setDataSource>	指定数据源
<sql:query>	运行SQL查询语句
<sql:update>	运行SQL更新语句
<sql:param>	将SQL语句中的参数设为指定值
<sql:dateParam>	将SQL语句中的日期参数设为指定的java.util.Date 对象值
<sql:transaction>	在共享数据库连接中提供嵌套的数据库行为元素，将所有语句以一个事务的形式来运行

XML 标签

JSTL XML标签库提供了创建和操作XML文档的标签。引用XML标签库的语法如下：

```
<%@ taglib prefix="x"
          uri="http://java.sun.com/jsp/jstl/xml" %>
```

在使用xml标签前，你必须将XML 和 XPath 的相关包拷贝至你的<Tomcat 安装目录>\lib下:

XercesImpl.jar:

下载地址： http://www.apache.org/dist/xerces/j/

xalan.jar:

下载地址： http://xml.apache.org/xalan-j/index.html

标签	描述
<x:out>	与<%= ... >,类似，不过只用于XPath表达式
<x:parse>	解析 XML 数据
<x:set>	设置XPath表达式
<x:if>	判断XPath表达式，若为真，则执行本体中的内容，否则跳过本体
<x:forEach>	迭代XML文档中的节点
<x:choose>	<x:when>和<x:otherwise>的父标签
<x:when>	<x:choose>的子标签，用来进行条件判断
<x:otherwise>	<x:choose>的子标签，当<x:when>判断为false时被执行
<x:transform>	将XSL转换应用在XML文档中
<x:param>	与<x:transform>共同使用，用于设置XSL样式表

JSTL函数

JSTL包含一系列标准函数，大部分是通用的字符串处理函数。引用JSTL函数库的语法如下：

```
<%@ taglib prefix="fn"
          uri="http://java.sun.com/jsp/jstl/functions" %>
```

函数	描述
fn:contains()	测试输入的字符串是否包含指定的子串
fn:containsIgnoreCase()	测试输入的字符串是否包含指定的子串，大小写不敏感
fn:endsWith()	测试输入的字符串是否以指定的后缀结尾
fn:escapeXml()	跳过可以作为XML标记的字符
fn:indexOf()	返回指定字符串在输入字符串中出现的位置
fn:join()	将数组中的元素合成一个字符串然后输出
fn:length()	返回字符串长度
fn:replace()	将输入字符串中指定的位置替换为指定的字符串然后返回
fn:split()	将字符串用指定的分隔符分隔然后组成一个子字符串数组并返回
fn:startsWith()	测试输入字符串是否以指定的前缀开始
fn:substring()	返回字符串的子集
fn:substringAfter()	返回字符串在指定子串之后的子集
fn:substringBefore()	返回字符串在指定子串之前的子集
fn:toLowerCase()	将字符串中的字符转为小写
fn:toUpperCase()	将字符串中的字符转为大写
fn:trim()	移除首位的空白符

JSP 连接数据库

本章节假设您已经对JDBC有一定的了解。在开始学习JSP数据库访问前，请确保JDBC环境已经正确配置。

首先，让我们按照下面的步骤来创建一个简单的表并插入几条简单的记录：

创建表

在数据库中创建一个Employees表，步骤如下：

步骤1：

打开CMD，然后进入数据库安装目录：

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

步骤2：

```
C:\Program Files\MySQL\bin>mysql -u root -p
Enter password: ********
mysql>
```

步骤3：

在TEST数据库中创建Employee表：

```
mysql> use TEST;
mysql> create table Employees
    (
     id int not null,
     age int not null,
     first varchar (255),
     last varchar (255)
    );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

插入数据记录

创建好Employee表后，往表中插入几条记录：

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

SELECT操作

接下来的这个例子告诉我们如何使用JSTL SQL标签来运行SQL SELECT语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>SELECT Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30
103	Sumit	Mittal	28

INSERT操作

这个例子告诉我们如何使用JSTL SQL标签来运行SQL INSERT语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>JINSERT Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<sql:update dataSource="${snapshot}" var="result">
INSERT INTO Employees VALUES (104, 2, 'Nuha', 'Ali');
</sql:update>

<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30
103	Sumit	Mittal	28
104	Nuha	Ali	2

DELETE操作

这个例子告诉我们如何使用JSTL SQL标签来运行SQL DELETE语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>DELETE Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<c:set var="empId" value="103"/>

<sql:update dataSource="${snapshot}" var="count">
  DELETE FROM Employees WHERE Id = ?
  <sql:param value="${empId}" />
</sql:update>

<sql:query dataSource="${snapshot}" var="result">
   SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30

UPDATE操作

这个例子告诉我们如何使用JSTL SQL标签来运行SQL UPDATE语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>DELETE Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<c:set var="empId" value="102"/>

<sql:update dataSource="${snapshot}" var="count">
  UPDATE Employees SET last = 'Ali'
  <sql:param value="${empId}" />
</sql:update>

<sql:query dataSource="${snapshot}" var="result">
   SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Ali	30

JSP XML 数据处理

当通过HTTP发送XML数据时，就有必要使用JSP来处理传入和流出的XML文档了，比如RSS文档。作为一个XML文档，它仅仅只是一堆文本而已，使用JSP创建XML文档并不比创建一个HTML文档难。

使用JSP发送XML

使用JSP发送XML内容就和发送HTML内容一样。唯一的不同就是您需要把页面的context属性设置为text/xml。要设置context属性，使用<%@page % >命令，就像这样：

```
<%@ page contentType="text/xml" %>
```

接下来这个例子向浏览器发送XML内容：

```
<%@ page contentType="text/xml" %>

<books>
   <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
   </book>
</books>
```

使用不同的浏览器来访问这个例子，看看这个例子所呈现的文档树。

在JSP中处理XML

在使用JSP处理XML之前，您需要将与XML 和XPath相关的两个库文件放在<Tomcat Installation Directory>\lib目录下：

- XercesImpl.jar：在这下载http://www.apache.org/dist/xerces/j/
- xalan.jar：在这下载http://xml.apache.org/xalan-j/index.html

books.xml文件:

```
<books>
<book>
  <name>Padam History</name>
  <author>ZARA</author>
  <price>100</price>
</book>
<book>
  <name>Great Mistry</name>
  <author>NUHA</author>
  <price>2000</price>
</book>
</books>
```

main.jsp文件：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<html>
<head>
  <title>JSTL x:parse Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:import var="bookInfo" url="http://localhost:8080/books.xml"/>

<x:parse xml="${bookInfo}" var="output"/>
<b>The title of the first book is</b>:
<x:out select="$output/books/book[1]/name" />
<br>
<b>The price of the second book</b>:
<x:out select="$output/books/book[2]/price" />

</body>
</html>
```

访问http://localhost:8080/main.jsp，运行结果如下：

```
BOOKS INFO:
The title of the first book is:Padam History
The price of the second book: 2000
```

使用JSP格式化XML

这个是XSLT样式表style.xsl文件：

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="html" indent="yes"/>

<xsl:template match="/">
  <html>
  <body>
   <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="books">
  <table border="1" width="100%">
    <xsl:for-each select="book">
      <tr>
        <td>
          <i><xsl:value-of select="name"/></i>
        </td>
        <td>
          <xsl:value-of select="author"/>
        </td>
        <td>
          <xsl:value-of select="price"/>
        </td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

这个是main.jsp文件：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<html>
<head>
  <title>JSTL x:transform Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:set var="xmltext">
  <books>
    <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
    </book>
    <book>
      <name>Great Mistry</name>
      <author>NUHA</author>
      <price>2000</price>
    </book>
  </books>
</c:set>

<c:import url="http://localhost:8080/style.xsl" var="xslt"/>
<x:transform xml="${xmltext}" xslt="${xslt}"/>

</body>
</html>
```

运行结果如下：

BOOKS INFO:

Padam History	ZARA	100
Great Mistry	NUHA	2000

更多关于使用JSTL处理XML的内容请查阅JSP标准标签库。

JSP JavaBean

JavaBean是特殊的Java类，使用J ava语言书写，并且遵守JavaBeans API规范。

接下来给出的是JavaBean与其它Java类相比而言独一无二的特征：

- 提供一个默认的无参构造函数。
- 需要被序列化并且实现了Serializable接口。
- 可能有一系列可读写属性。
- 可能有一系列的"getter"或"setter"方法。

JavaBeans属性

一个JavaBean对象的属性应该是可访问的。这个属性可以是任意合法的Java数据类型，包括自定义Java类。

一个JavaBean对象的属性可以是可读写，或只读，或只写。JavaBean对象的属性通过JavaBean实现类中提供的两个方法来访问：

方法	描述
get**PropertyName**()	举例来说，如果属性的名称为myName，那么这个方法的名字就要写成getMyName()来读取这个属性。这个方法也称为访问器。
set**PropertyName**()	举例来说，如果属性的名称为myName，那么这个方法的名字就要写成setMyName()来写入这个属性。这个方法也称为写入器。

一个只读的属性只提供getPropertyName()方法，一个只写的属性只提供setPropertyName()方法。

JavaBeans程序示例

这是StudentBean.java文件：

```
package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable
{
   private String firstName = null;
   private String lastName = null;
   private int age = 0;

   public StudentsBean() {
   }
   public String getFirstName(){
      return firstName;
   }
   public String getLastName(){
      return lastName;
   }
   public int getAge(){
      return age;
   }
   public void setFirstName(String firstName){
      this.firstName = firstName;
   }
   public void setLastName(String lastName){
      this.lastName = lastName;
   }
   public void setAge(Integer age){
      this.age = age;
   }
}
```

编译StudentBean.java文件，在本章最后的例子中将会使用到它。

访问JavaBeans

<jsp:useBean>标签可以在JSP中声明一个JavaBean，然后使用。声明后，JavaBean对象就成了脚本变量，可以通过脚本元素或其他自定义标签来访问。<jsp:useBean>标签的语法格式如下：

```
<jsp:useBean id="bean's name" scope="bean's scope" typeSpec/>
```

其中，根据具体情况，scope的值可以是page，request，session或application。id值可任意只要不和同一JSP文件中其它<jsp:useBean>中id值一样就行了。

接下来给出的是<jsp:useBean>标签的一个简单的用法：

```
<html>
<head>
<title>useBean Example</title>
</head>
<body>

<jsp:useBean id="date" class="java.util.Date" />
<p>The date/time is <%= date %>

</body>
</html>
```

它将会产生如下结果：

```
The date/time is Thu Sep 30 11:18:11 GST 2013
```

访问JavaBeans对象的属性

在<jsp:useBean>标签主体中使用<jsp:getProperty/>标签来调用getter方法，使用<jsp:setProperty/>标签来调用setter方法，语法格式如下：

```
<jsp:useBean id="id" class="bean's class" scope="bean's scope">
   <jsp:setProperty name="bean's id" property="property name"
                    value="value"/>
   <jsp:getProperty name="bean's id" property="property name"/>
   ...........
</jsp:useBean>
```

name属性指的是Bean的id属性。property属性指的是想要调用的getter或setter方法。

接下来给出使用以上语法进行属性访问的一个简单例子：

```
<html>
<head>
<title>get and set properties Example</title>
</head>
<body>

<jsp:useBean id="students"
                    class="com.tutorialspoint.StudentsBean">
   <jsp:setProperty name="students" property="firstName"
                    value="Zara"/>
   <jsp:setProperty name="students" property="lastName"
                    value="Ali"/>
   <jsp:setProperty name="students" property="age"
                    value="10"/>
</jsp:useBean>

<p>Student First Name:
   <jsp:getProperty name="students" property="firstName"/>
</p>
<p>Student Last Name:
   <jsp:getProperty name="students" property="lastName"/>
</p>
<p>Student Age:
   <jsp:getProperty name="students" property="age"/>
</p>

</body>
</html>
```

将StudentBean.class加入CLASSPATH环境变量中，然后访问以上JSP，运行结果如下：

```
Student First Name: Zara

Student Last Name: Ali

Student Age: 10
```

JSP 自定义标签

自定义标签是用户定义的JSP语言元素。当JSP页面包含一个自定义标签时将被转化为servlet，标签转化为对被 称为tag handler的对象的操作，即当servlet执行时Web container调用那些操作。

JSP标签扩展可以让你创建新的标签并且可以直接插入到一个JSP页面。 JSP 2.0规范中引入Simple Tag Handlers来编写这些自定义标记。

你可以继承SimpleTagSupport类并重写的doTag()方法来开发一个最简单的自定义标签。

创建"Hello"标签

接下来，我们想创建一个自定义标签叫作<ex:Hello>，标签格式为：

```
<ex:Hello />
```

要创建自定义的JSP标签，你首先必须创建处理标签的Java类。所以，让我们创建一个HelloTag类，如下所示：

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.println("Hello Custom Tag!");
  }
}
```

以下代码重写了doTag()方法，方法中使用了getJspContext()方法来获取当前的JspContext对象，并将"Hello Custom Tag!"传递给JspWriter对象。

编译以上类，并将其复制到环境变量CLASSPATH目录中。最后创建如下标签库：<Tomcat安装目录>webapps\ROOT\WEB-INF\custom.tld。

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

接下来，我们就可以在JSP文件中使用Hello标签：

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello/>
  </body>
</html>
```

以上程序输出结果为：

```
Hello Custom Tag!
```

访问标签体

你可以像标准标签库一样在标签中包含消息内容。如我们要在我们自定义的Hello中包含内容，格式如下：

```
<ex:Hello>
   This is message body
</ex:Hello>
```

我们可以修改标签处理类文件，代码如下：

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

   StringWriter sw = new StringWriter();
   public void doTag()
      throws JspException, IOException
    {
       getJspBody().invoke(sw);
       getJspContext().getOut().println(sw.toString());
    }

}
```

接下来我们需要修改TLD文件，如下所示：

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

现在我们可以在JSP使用修改后的标签，如下所示:

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello>
        This is message body
    </ex:Hello>
  </body>
</html>
```

以上程序输出结果如下所示：

```
This is message body
```

自定义标签属性

你可以在自定义标准中设置各种属性，要接收属性，值自定义标签类必须实现setter方法，JavaBean 中的setter方法如下所示：

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

   private String message;

   public void setMessage(String msg) {
      this.message = msg;
   }

   StringWriter sw = new StringWriter();

   public void doTag()
      throws JspException, IOException
    {
       if (message != null) {
          /* 从属性中使用消息 */
          JspWriter out = getJspContext().getOut();
          out.println( message );
       }
       else {
          /* 从内容体中使用消息 */
          getJspBody().invoke(sw);
          getJspContext().getOut().println(sw.toString());
       }
   }

}
```

属性的名称是"message"，所以setter方法??是的setMessage()。现在让我们在TLD文件中使用的<attribute>元素添加此属性：

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
       <name>message</name>
    </attribute>
  </tag>
</taglib>
```

现在我们就可以在JSP文件中使用message属性了，如下所示：

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello message="This is custom tag" />
  </body>
</html>
```

以上实例数据输出结果为：

```
This is custom tag
```

你还可以包含以下属性：

属性	描述
name	定义属性的名称。每个标签的是属性名称必须是唯一的。
required	指定属性是否是必须的或者可选的,如果设置为false为可选。
rtexprvalue	声明在运行表达式时，标签属性是否有效。
type	定义该属性的Java类类型 。默认指定为 **String**
description	描述信息
fragment	如果声明了该属性,属性值将被视为一个 **JspFragment**。

以下是指定相关的属性实例：

```
.....
    <attribute>
      <name>attribute_name</name>
      <required>false</required>
      <type>java.util.Date</type>
      <fragment>false</fragment>
    </attribute>
.....
```

如果你使用了两个属性，修改TLD文件，如下所示：

```
.....
    <attribute>
      <name>attribute_name1</name>
      <required>false</required>
      <type>java.util.Boolean</type>
      <fragment>false</fragment>
    </attribute>
    <attribute>
      <name>attribute_name2</name>
      <required>true</required>
      <type>java.util.Date</type>
    </attribute>
.....
```

JSP 表达式语言

JSP表达式语言（EL）使得访问存储在JavaBean中的数据变得非常简单。JSP EL既可以用来创建算术表达式也可以用来创建逻辑表达式。在JSP EL表达式内可以使用整型数，浮点数，字符串，常量true、false，还有null。

一个简单的语法

典型的，当您需要在JSP标签中指定一个属性值时，只需要简单地使用字符串即可：

```
<jsp:setProperty name="box" property="perimeter" value="100"/>
```

JSP EL允许您指定一个表达式来表示属性值。一个简单的表达式语法如下：

```
${expr}
```

其中，expr指的是表达式。在JSP EL中通用的操作符是"."和"[]"。这两个操作符允许您通过内嵌的JSP对象访问各种各样的JavaBean属性。

举例来说，上面的<jsp:setProperty>标签可以使用表达式语言改写成如下形式：

```
<jsp:setProperty name="box" property="perimeter"
                 value="${2*box.width+2*box.height}"/>
```

当JSP编译器在属性中见到"${}"格式后，它会产生代码来计算这个表达式，并且产生一个替代品来代替表达式的值。

您也可以在标签的模板文本中使用表达式语言。比如<jsp:text>标签简单地将其主体中的文本插入到JSP输出中：

```
<jsp:text>
<h1>Hello JSP!</h1>
</jsp:text>
```

现在，在<jsp:text>标签主体中使用表达式，就像这样：

```
<jsp:text>
Box Perimeter is: ${2*box.width + 2*box.height}
</jsp:text>
```

在EL表达式中可以使用圆括号来组织子表达式。比如${(1 + 2) *3}*等于*9*，但是*${1 + (2* 3)}* 等于7。

想要停用对EL表达式的评估的话，需要使用page指令将isELIgnored属性值设为true：

```
<%@ page isELIgnored ="true|false" %>
```

这样，EL表达式就会被忽略。若设为false，则容器将会计算EL表达式。

EL中的基础操作符

EL表达式支持大部分Java所提供的算术和逻辑操作符：

操作符	描述
.	访问一个Bean属性或者一个映射条目
[]	访问一个数组或者链表的元素
( )	组织一个子表达式以改变优先级
+	加
-	减或负
*	乘
/ or div	除
% or mod	取模
== or eq	测试是否相等
!= or ne	测试是否不等
< or lt	测试是否小于
> or gt	测试是否大于
<= or le	测试是否小于等于
>= or gt	测试是否大于等于
&& or and	测试逻辑与
|| or or	测试逻辑或
! or not	测试取反
empty	测试是否空值

JSP EL中的函数

JSP EL允许您在表达式中使用函数。这些函数必须被定义在自定义标签库中。函数的使用语法如下：

```
${ns:func(param1, param2, ...)}
```

ns指的是命名空间（namespace），func指的是函数的名称，param1指的是第一个参数，param2指的是第二个参数，以此类推。比如，有函数fn:length，在JSTL库中定义，可以像下面这样来获取一个字符串的长度：

```
${fn:length("Get my length")}
```

要使用任何标签库中的函数，您需要将这些库安装在服务器中，然后使用<taglib>标签在JSP文件中包含这些库。

JSP EL隐含对象

JSP EL支持下表列出的隐含对象：

隐含对象	描述
pageScope	page 作用域
requestScope	request 作用域
sessionScope	session 作用域
applicationScope	application 作用域
param	Request 对象的参数，字符串
paramValues	Request对象的参数，字符串集合
header	HTTP 信息头，字符串
headerValues	HTTP 信息头，字符串集合
initParam	上下文初始化参数
cookie	Cookie值
pageContext	当前页面的pageContext

您可以在表达式中使用这些对象，就像使用变量一样。接下来会给出几个例子来更好的理解这个概念。

pageContext对象

pageContext对象是JSP中pageContext对象的引用。通过pageContext对象，您可以访问request对象。比如，访问request对象传入的查询字符串，就像这样：

```
${pageContext.request.queryString}
```

Scope对象

pageScope，requestScope，sessionScope，applicationScope变量用来访问存储在各个作用域层次的变量。

举例来说，如果您需要显式访问在applicationScope层的box变量，可以这样来访问：applicationScope.box。

param和paramValues对象

param和paramValues对象用来访问参数值，通过使用request.getParameter方法和request.getParameterValues方法。

举例来说，访问一个名为order的参数，可以这样使用表达式：${param.order}，或者${param["order"]}。

接下来的例子表明了如何访问request中的username参数：

```
<%@ page import="java.io.*,java.util.*" %>
<%
    String title = "Accessing Request Param";
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>${param["username"]}</p>
</div>
</body>
</html>
```

param对象返回单一的字符串，而paramValues对象则返回一个字符串数组。

header和headerValues对象

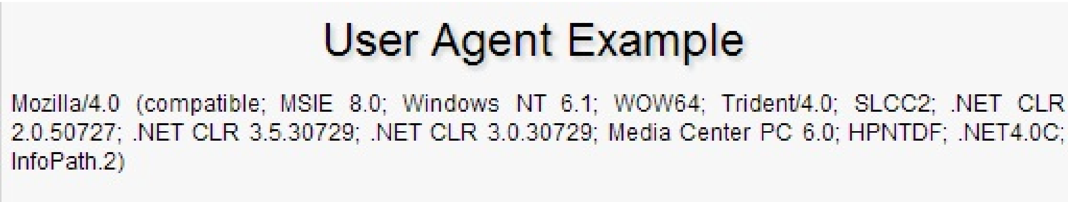
header和headerValues对象用来访问信息头，通过使用 request.getHeader方法和request.getHeaders方法。

举例来说，要访问一个名为user-agent的信息头，可以这样使用表达式：${header.user-agent}，或者${header["user-agent"]}。

接下来的例子表明了如何访问user-agent信息头：

```
<%@ page import="java.io.*,java.util.*" %>
<%
    String title = "User Agent Example";
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>${header["user-agent"]}</p>
</div>
</body>
</html>
```

运行结果如下：



header对象返回单一值，而headerValues则返回一个字符串数组。

JSP 异常处理

当编写JSP程序的时候，程序员可能会遗漏一些BUG，这些BUG可能会出现在程序的任何地方。JSP代码中通常有以下几类异常:

- 检查型异常:检查型异常就是一个典型的用户错误或者一个程序员无法预见的错误。举例来说，如果一个文件将要被打开，但是无法找到这个文件，则一个异常被抛出。这些异常不能再编译期被简单地忽略。
- 运行时异常:一个运行时异常可能已经被程序员避免，这种异常在编译期将会被忽略。
- 错误:这里没有异常，但问题是它超出了用户或者程序员的控制范围。错误通常会在代码中被忽略，您几乎不能拿它怎么样。举例来或，栈溢出错误。这些错误都会在编译期被忽略。

本节将会给出几个简单而优雅的方式来处理运行时异常和错误。

使用Exception对象

exception对象是Throwable子类的一个实例，只在错误页面中可用。下表列出了Throwable类中一些重要的方法:

方法	描述
public String getMessage()	返回异常的信息。这个信息在Throwable构造函数中被初始化
public ThrowablegetCause()	返回引起异常的原因，类型为Throwable对象
public String toString()	返回类名
public void printStackTrace()	将异常栈轨迹输出至System.err
public StackTraceElement [] getStackTrace()	以栈轨迹元素数组的形式返回异常栈轨迹
public ThrowablefillInStackTrace()	使用当前栈轨迹填充Throwable对象

JSP提供了可选项来为每个JSP页面指定错误页面。无论何时页面抛出了异常，JSP容器都会自动地调用错误页面。

接下来的例子为main.jsp指定了一个错误页面。使用<%@page errorPage="XXXXX"%>指令指定一个错误页面。

```
<%@ page errorPage="ShowError.jsp" %>

<html>
<head>
   <title>Error Handling Example</title>
</head>
<body>
<%
   // Throw an exception to invoke the error page
   int x = 1;
   if (x == 1)
   {
      throw new RuntimeException("Error condition!!!");
   }
%>
</body>
</html>
```

现在，编写ShowError.jsp文件如下:

```
<%@ page isErrorPage="true" %>
<html>
<head>
<title>Show Error Page</title>
</head>
<body>
<h1>Opps...</h1>
<p>Sorry, an error occurred.</p>
<p>Here is the exception stack trace: </p>
<pre>
<% exception.printStackTrace(response.getWriter()); %>
```

注意到，ShowError.jsp文件使用了<%@page isErrorPage="true"%>指令，这个指令告诉JSP编译器需要产生一个异常实例变量。

现在试着访问main.jsp页面，它将会产生如下结果:

```
java.lang.RuntimeException: Error condition!!!
......

Opps...
Sorry, an error occurred.

Here is the exception stack trace:
```

在错误页面中使用JSTL标签

可以利用JSTL标签来编写错误页面ShowError.jsp。这个例子中的代码与上例代码的逻辑几乎一样，但是本例的代码有更好的结构，并且能够提供更多信息:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page isErrorPage="true" %>
<html>
<head>
<title>Show Error Page</title>
</head>
<body>
<h1>Opps...</h1>
<table width="100%" border="1">
<tr valign="top">
<td width="40%"><b>Error:</b></td>
<td>${pageContext.exception}</td>
</tr>
<tr valign="top">
<td><b>URI:</b></td>
<td>${pageContext.errorData.requestURI}</td>
</tr>
<tr valign="top">
<td><b>Status code:</b></td>
<td>${pageContext.errorData.statusCode}</td>
</tr>
<tr valign="top">
<td><b>Stack trace:</b></td>
<td>
<c:forEach var="trace"
         items="${pageContext.exception.stackTrace}">
<p>${trace}</p>
</c:forEach>
</td>
</tr>
</table>
</body>
</html>
```

运行结果如下:

Opps...

Error:	java.lang.RuntimeException: Error condition!!!
URI:	/main.jsp
Status code:	500
Stack trace:	org.apache.jsp.main_jsp._jspService(main_jsp.java:65) org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:68) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) ....

使用 try…catch块

如果您想要将异常处理放在一个页面中，并且对不同的异常进行不同的处理，那么您就需要使用try…catch块了。

接下来的这个例子显示了如何使用try...catch块，将这些代码放在main.jsp中:

```
<html>
<head>
   <title>Try...Catch Example</title>
</head>
<body>
<%
   try{
      int i = 1;
      i = i / 0;
      out.println("The answer is " + i);
   }
   catch (Exception e){
      out.println("An exception occurred: " + e.getMessage());
   }
%>
</body>
</html>
```

试着访问main.jsp，它将会产生如下结果:

```
An exception occurred: / by zero
```

JSP 调试

要测试/调试一个JSP或servlet程序总是那么的难。JSP和Servlets程序趋向于牵涉到大量客户端/服务器之间的交互，这很有可能会产生错误，并且很难重现出错的环境。

接下来将会给出一些小技巧和小建议，来帮助您调试程序。

使用System.out.println()

System.out.println()可以很方便地标记一段代码是否被执行。当然，我们也可以打印出各种各样的值。此外：

- 自从System对象成为Java核心对象后，它便可以使用在任何地方而不用引入额外的类。使用范围包括Servlets，JSP，RMI，EJB's，Beans，类和独立应用。
- 与在断点处停止运行相比，用System.out进行输出不会对应用程序的运行流程造成重大的影响，这个特点在定时机制非常重要的应用程序中就显得非常有用了。

接下来给出了使用System.out.println()的语法：

```
System.out.println("Debugging message");
```

这是一个使用System.out.print()的简单例子：

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>System.out.println</title></head>
<body>
<c:forEach var="counter" begin="1" end="10" step="1" >
   <c:out value="${counter-5}"/></br>
   <% System.out.println( "counter= " +
                     pageContext.findAttribute("counter") ); %>
</c:forEach>
</body>
</html>
```

现在，如果运行上面的例子的话，它将会产生如下的结果：

```
-4
-3
-2
-1
0
1
2
3
4
5
```

如果使用的是Tomcat服务器，您就能够在logs目录下的stdout.log文件中发现多出了如下内容：

```
counter=1
counter=2
counter=3
counter=4
counter=5
counter=6
counter=7
counter=8
counter=9
counter=10
```

使用这种方法可以将变量和其它的信息输出至系统日志中，用来分析并找出造成问题的深层次原因。

使用JDB Logger

J2SE日志框架可为任何运行在JVM中的类提供日志记录服务。因此我们可以利用这个框架来记录任何信息。

让我们来重写以上代码，使用JDK中的 logger API：

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page import="java.util.logging.Logger" %>

<html>
<head><title>Logger.info</title></head>
<body>
<% Logger logger=Logger.getLogger(this.getClass().getName());%>

<c:forEach var="counter" begin="1" end="10" step="1" >
   <c:set var="myCount" value="${counter-5}" />
   <c:out value="${myCount}"/></br>
   <% String message = "counter="
                  + pageContext.findAttribute("counter")
                  + " myCount="
                  + pageContext.findAttribute("myCount");
                  logger.info( message );
   %>
</c:forEach>
</body>
</html>
```

它的运行结果与先前的类似，但是，它可以获得额外的信息输出至stdout.log文件中。在这我们使用了logger中的info方法。下面我们给出stdout.log文件中的一个快照：

```
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=1 myCount=-4
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=2 myCount=-3
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=3 myCount=-2
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=4 myCount=-1
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=5 myCount=0
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=6 myCount=1
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=7 myCount=2
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=8 myCount=3
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=9 myCount=4
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=10 myCount=5
```

消息可以使用各种优先级发送，通过使用sever()，warning()，info()，config()，fine()，finer()，finest()方法。finest()方法用来记录最好的信息，而sever()方法用来记录最严重的信息。

使用Log4J 框架来将消息记录在不同的文件中，这些消息基于严重程度和重要性来进行分类。

调试工具

NetBeans是树形结构，是开源的Java综合开发环境，支持开发独立的Java应用程序和网络应用程序，同时也支持JSP调试。

NetBeans支持如下几个基本的调试功能：

- 断点
- 单步跟踪
- 观察点

详细的信息可以查看NetBeans使用手册。

使用JDB Debugger

可以在JSP和servlets中使用jdb命令来进行调试，就像调试普通的应用程序一样。

通常，我们直接调试sun.servlet.http.HttpServer 对象来查看HttpServer在响应HTTP请求时执行JSP/Servlets的情况。这与调试applets非常相似。不同之处在于，applets程序实际调试的是sun.applet.AppletViewer。

大部分调试器在调试applets时都能够自动忽略掉一些细节，因为它知道如何调试applets。如果想要将调试对象转移到JSP身上，就需要做好以下两点：

- 设置调试器的classpath，让它能够找到sun.servlet.http.Http-Server 和相关的类。
- 设置调试器的classpath，让它能够找到您的JSP文件和相关的类。

设置好classpath后，开始调试sun.servlet.http.Http-Server 。您可以在JSP文件的任意地方设置断点，只要你喜欢，然后使用浏览器发送一个请求给服务器就应该可以看见程序停在了断点处。

使用注释

程序中的注释在很多方面都对程序的调试起到一定的帮助作用。注释可以用在调试程序的很多方面中。

JSP使用Java注释。如果一个BUG消失了，就请仔细查看您刚注释过的代码，通常都能找出原因。

客户端和服务器的头模块

有时候，当JSP没有按照预定的方式运行时，查看未加工的HTTP请求和响应也是很有用的。如果对HTTP的结构很熟悉的话，您可以直接观察request和response然后看看这些头模块到底怎么了。

重要调试技巧

这里我们再透露两个调试JSP的小技巧：

- 使用浏览器显示原始的页面内容，用来区分是否是格式问题。这个选项通常在View菜单下。
- 确保浏览器在强制重新载入页面时没有捕获先前的request输出。若使用的是Netscape Navigator浏览器，则用Shift-Reload；若使用的是IE浏览器，则用Shift-Refresh。

JSP 国际化

在开始前，需要解释几个重要的概念：

- 国际化（i18n）：表明一个页面根据访问者的语言或国家来呈现不同的翻译版本。
- 本地化（l10n）：向网站添加资源，以使它适应不同的地区和文化。比如网站的印度语版本。
- 区域：这是一个特定的区域或文化，通常认为是一个语言标志和国家标志通过下划线连接起来。比如"en_US"代表美国英语地区。

如果想要建立一个全球化的网站，就需要关心一系列项目。本章将会详细告诉您如何处理国际化问题，并给出了一些例子来加深理解。

JSP容器能够根据request的locale属性来提供正确地页面版本。接下来给出了如何通过request对象来获得Locale对象的语法：

```
java.util.Locale request.getLocale()
```

检测Locale

下表列举出了Locale对象中比较重要的方法，用于检测request对象的地区，语言，和区域。所有这些方法都会在浏览器中显示国家名称和语言名称：

方法	描述
String getCountry()	返回国家/地区码的英文大写，或 ISO 3166 2-letter 格式的区域
String getDisplayCountry()	返回要显示给用户的国家名称
String getLanguage()	返回语言码的英文小写，或ISO 639 格式的区域
String getDisplayLanguage()	返回要给用户看的语言名称
String getISO3Country()	返回国家名称的3字母缩写
String getISO3Language()	返回语言名称的3字母缩写

实例演示

这个例子告诉我们如何在JSP中显示语言和国家：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%
   //获取客户端本地化信息
   Locale locale = request.getLocale();
   String language = locale.getLanguage();
   String country = locale.getCountry();
%>
<html>
<head>
<title>Detecting Locale</title>
</head>
<body>
<center>
<h1>Detecting Locale</h1>
</center>
<p align="center">
<%
   out.println("Language : " + language  + "<br />");
   out.println("Country  : " + country   + "<br />");
%>
</p>
</body>
</html>
```

语言设置

JSP可以使用西欧语言来输出一个页面，比如英语，西班牙语，德语，法语，意大利语等等。由此可见，设置Content-Language信息头来正确显示所有字符是很重要的。

第二点就是，需要使用HTML字符实体来显示特殊字符，比如"ñ" 代表的是"?"，"¡"代表的是"?" ：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%
    // Set response content type
    response.setContentType("text/html");
    // Set spanish language code.
    response.setHeader("Content-Language", "es");
    String title = "En Espa?ol";

%>
<html>
<head>
<title><%  out.print(title); %></title>
</head>
<body>
<center>
<h1><%  out.print(title); %></h1>
</center>
<div align="center">
<p>En Espa?ol</p>
<p>?Hola Mundo!</p>
</div>
</body>
</html>
```

区域特定日期

可以使用java.text.DateFormat类和它的静态方法getDateTimeInstance()来格式化日期和时间。接下来的这个例子显示了如何根据指定的区域来格式化日期和时间：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.DateFormat,java.util.Date" %>

<%
    String title = "Locale Specific Dates";
    //Get the client's Locale
    Locale locale = request.getLocale( );
    String date = DateFormat.getDateTimeInstance(
                                  DateFormat.FULL,
                                  DateFormat.SHORT,
                                  locale).format(new Date( ));
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Local Date: <%  out.print(date); %></p>
</div>
</body>
</html>
```

区域特定货币

可以使用java.text.NumberFormat类和它的静态方法getCurrencyInstance()来格式化数字。比如在区域特定货币中的long型和double型。接下来的例子显示了如何根据指定的区域来格式化货币：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.NumberFormat,java.util.Date" %>

<%
    String title = "Locale Specific Currency";
    //Get the client's Locale
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getCurrencyInstance(locale);
    String formattedCurr = nft.format(1000000);
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Formatted Currency: <%  out.print(formattedCurr); %></p>
</div>
</body>
</html>
```

区域特定百分比

可以使用java.text.NumberFormat类和它的静态方法getPercentInstance()来格式化百分比。接下来的例子告诉我们如何根据指定的区域来格式化百分比：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.NumberFormat,java.util.Date" %>

<%
    String title = "Locale Specific Percentage";
    //Get the client's Locale
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getPercentInstance(locale);
    String formattedPerc = nft.format(0.51);
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Formatted Percentage: <%  out.print(formattedPerc); %></p>
</div>
</body>
</html>
```

W3School C# 教程

来源：C#教程

整理：飞龙

C# 基础

C# 简介

C# 是一个现代的、通用的、面向对象的编程语言，它是由微软（Microsoft）开发的，由 Ecma 和 ISO 核准认可的。

C# 是由 Anders Hejlsberg 和他的团队在 .Net 框架开发期间开发的。

C# 是专为公共语言基础结构（CLI）设计的。CLI 由可执行代码和运行时环境组成，允许在不同的计算机平台和体系结构上使用各种高级语言。

下面列出了 C# 成为一种广泛应用的专业语言的原因：

- 现代的、通用的编程语言。
- 面向对象。
- 面向组件。
- 容易学习。
- 结构化语言。
- 它产生高效率的程序。
- 它可以在多种计算机平台上编译。
- .Net 框架的一部分。

C# 强大的编程功能

虽然 C# 的构想十分接近于传统高级语言 C 和 C++，是一门面向对象的编程语言，但是它与 Java 非常相似，有许多强大的编程功能，因此得到广大程序员的亲睐。

下面列出 C# 一些重要的功能：

- 布尔条件（Boolean Conditions）
- 自动垃圾回收（Automatic Garbage Collection）
- 标准库（Standard Library）
- 组件版本（Assembly Versioning）
- 属性（Properties）和事件（Events）
- 委托（Delegates）和事件管理（Events Management）
- 易于使用的泛型（Generics）
- 索引器（Indexers）
- 条件编译（Conditional Compilation）
- 简单的多线程（Multithreading）
- LINQ 和 Lambda 表达式
- 集成 Windows

C# 环境

在这一章中，我们将讨论创建 C# 编程所需的工具。我们已经提到 C# 是 .Net 框架的一部分，且用于编写 .Net 应用程序。因此，在讨论运行 C# 程序的可用工具之前，让我们先了解一下 C# 与 .Net 框架之间的关系。

.Net 框架（.Net Framework）

.Net 框架是一个创新的平台，能帮您编写出下面类型的应用程序：

- Windows 应用程序
- Web 应用程序
- Web 服务

.Net 框架应用程序是多平台的应用程序。框架的设计方式使它适用于下列各种语言：C#、C++、Visual Basic、Jscript、COBOL 等等。所有这些语言可以访问框架，彼此之间也可以互相交互。

.Net 框架由一个巨大的代码库组成，用于 C# 等客户端语言。下面列出一些 .Net 框架的组件：

- 公共语言运行库（Common Language Runtime - CLR）
- .Net 框架类库（.Net Framework Class Library）
- 公共语言规范（Common Language Specification）
- 通用类型系统（Common Type System）
- 元数据（Metadata）和组件（Assemblies）
- Windows 窗体（Windows Forms）
- ASP.Net 和 ASP.Net AJAX
- ADO.Net
- Windows 工作流基础（Windows Workflow Foundation - WF）
- Windows 显示基础（Windows Presentation Foundation）
- Windows 通信基础（Windows Communication Foundation - WCF）
- LINQ

如需了解每个组件的详细信息，请参阅微软（Microsoft）的文档。

C# 的集成开发环境（Integrated Development Environment - IDE）

微软（Microsoft）提供了下列用于 C# 编程的开发工具：

- Visual Studio 2010 (VS)
- Visual C# 2010 Express (VCE)
- Visual Web Developer

后面两个是免费使用的，可从微软官方网址下载。使用这些工具，您可以编写各种 C# 程序，从简单的命令行应用程序到更复杂的应用程序。您也可以使用基本的文本编辑器（比如 Notepad）编写 C# 源代码文件，并使用命令行编译器（.NET 框架的一部分）编译代码为组件。

Visual C# Express 和 Visual Web Developer Express 版本是 Visual Studio 的定制版本，且具有相同的外观和感观。它们保留 Visual Studio 的大部分功能。在本教程中，我们使用的是 Visual C# 2010 Express。

您可以从 Microsoft Visual Studio 上进行下载。它会自动安装在您的机器上。请注意，您需要一个可用的网络连接来完成速成版的安装。

在 Linux 或 Mac OS 上编写 C# 程序

虽然 .NET 框架是运行在 Windows 操作系统上，但是也有一些运行于其它操作系统上的版本可供选择。**Mono** 是 .NET 框架的一个开源版本，它包含了一个 C# 编译器，且可运行于多种操作系统上，比如各种版本的 Linux 和 Mac OS。如需了解更多详情，请访问 Go Mono。

Mono 的目的不仅仅是跨平台地运行微软 .NET 应用程序，而且也为 Linux 开发者提供了更好的开发工具。Mono 可运行在多种操作系统上，包括 Android、BSD、iOS、Linux、OS X、Windows、Solaris 和 UNIX。

C# 程序结构

在我们学习 C# 编程语言的基础构件块之前，让我们先看一下 C# 的最小的程序结构，以便作为接下来章节的参考。

C# Hello World 实例

一个 C# 程序主要包括以下部分：

- 命名空间声明（Namespace declaration）
- 一个 class
- Class 方法
- Class 属性
- 一个 Main 方法
- 语句（Statements）& 表达式（Expressions）
- 注释

让我们看一个可以打印出 "Hello World" 的简单的代码：

```
using System;
namespace HelloWorldApplication
{
   class HelloWorld
   {
      static void Main(string[] args)
      {
         /* 我的第一个 C# 程序*/
         Console.WriteLine("Hello World");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello World
```

让我们看一下上面程序的各个部分：

- 程序的第一行 **using System;** - **using** 关键字用于在程序中包含 **System** 命名空间。 一个程序一般有多个 **using** 语句。
- 下一行是 **namespace** 声明。一个 **namespace** 是一系列的类。*HelloWorldApplication* 命名空间包含了类 *HelloWorld*。
- 下一行是 **class** 声明。类 *HelloWorld* 包含了程序使用的数据和方法声明。类一般包含多个方法。方法定义了类的行为。在这里，*HelloWorld* 类只有一个 **Main** 方法。

- 下一行定义了 **Main** 方法，是所有 C# 程序的 入口点。**Main** 方法说明当执行时 类将做什么动作。
- 下一行 /.../ 将会被编译器忽略，且它会在程序中添加额外的 注释。
- Main 方法通过语句 **Console.WriteLine("Hello World");** 指定了它的行为。

  WriteLine 是一个定义在 *System* 命名空间中的 *Console* 类的一个方法。该语句会在屏幕上显示消息 "Hello, World!"。

- 最后一行 **Console.ReadKey();** 是针对 VS.NET 用户的。这使得程序会等待一个按键的动作，防止程序从 Visual Studio .NET 启动时屏幕会快速运行并关闭。

以下几点值得注意：

- C# 是大小写敏感的。
- 所有的语句和表达式必须以分号（;）结尾。
- 程序的执行从 Main 方法开始。
- 与 Java 不同的是，文件名可以不同于类的名称。

编译 & 执行 C# 程序

如果您使用 Visual Studio.Net 编译和执行 C# 程序，请按下面的步骤进行：

- 启动 Visual Studio。
- 在菜单栏上，选择 File -> New -> Project。
- 从模板中选择 Visual C#，然后选择 Windows。
- 选择 Console Application。
- 为您的项目制定一个名称，然后点击 OK 按钮。
- 新项目会出现在解决方案资源管理器（Solution Explorer）中。
- 在代码编辑器（Code Editor）中编写代码。
- 点击 Run 按钮或者按下 F5 键来运行程序。会出现一个命令提示符窗口（Command Prompt window），显示 Hello World。

您也可以使用命令行代替 Visual Studio IDE 来编译 C# 程序：

- 打开一个文本编辑器，添加上面提到的代码。
- 保存文件为 **helloworld.cs**。
- 打开命令提示符工具，定位到文件所保存的目录。
- 键入 **csc helloworld.cs** 并按下 enter 键来编译代码。
- 如果代码没有错误，命令提示符会进入下一行，并生成 **helloworld.exe** 可执行文件。
- 接下来，键入 **helloworld** 来执行程序。
- 您将看到 "Hello World" 打印在屏幕上。

C# 基本语法

C# 是一种面向对象的编程语言。在面向对象的程序设计方法中，程序由各种相互交互的对象组成。相同种类的对象通常具有相同的类型，或者说，是在相同的 class 中。

例如，以 Rectangle（矩形）对象为例。它具有 length 和 width 属性。根据设计，它可能需要接受这些属性值、计算面积和显示细节。

让我们来看看一个 Rectangle（矩形）类的实现，并借此讨论 C# 的基本语法：

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        // 成员变量
        double length;
        double width;
        public void Acceptdetails()
        {
            length = 4.5;
            width = 3.5;
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }

    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Length: 4.5
Width: 3.5
Area: 15.75
```

using 关键字

在任何 C# 程序中的第一条语句都是：

```
using System;
```

using 关键字用于在程序中包含命名空间。一个程序可以包含多个 using 语句。

class 关键字

class 关键字用于声明一个类。

C# 中的注释

注释是用于解释代码。编译器会忽略注释的条目。在 C# 程序中，多行注释以 / 开始，并以字符 / 终止，如下所示：

```
/* This program demonstrates
The basic syntax of C# programming
Language */
```

单行注释是用 '//' 符号表示。例如：

```
}//end class Rectangle
```

成员变量

变量是类的属性或数据成员，用于存储数据。在上面的程序中，*Rectangle* 类有两个成员变量，名为 *length* 和 *width*。

成员函数

函数是一系列执行指定任务的语句。类的成员函数是在类内声明的。我们举例的类 Rectangle 包含了三个成员函数：*AcceptDetails*、*GetArea* 和 *Display*。

实例化一个类

在上面的程序中，类 *ExecuteRectangle* 是一个包含 *Main()* 方法和实例化 *Rectangle* 类的类。

标识符

标识符是用来识别类、变量、函数或任何其它用户定义的项目。在 C# 中，类的命名必须遵循如下基本规则：

- 标识符必须以字母开头，后面可以跟一系列的字母、数字（0 - 9）或下划线（ _ ）。标识符中的第一个字符不能是数字。
- 标识符必须不包含任何嵌入的空格或符号，比如 ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / \。但是，可以使用下划线（ _ ）。
- 标识符不能是 C# 关键字。

C# 关键字

关键字是 C# 编译器预定义的保留字。这些关键字不能用作标识符，但是，如果您想使用这些关键字作为标识符，可以在关键字前面加上 @ 字符作为前缀。

在 C# 中，有些标识符在代码的上下文中有特殊的意义，如 get 和 set，这些被称为上下文关键字（contextual keywords）。

下表列出了 C# 中的保留关键字（Reserved Keywords）和上下文关键字（Contextual Keywords）：

保留关键字						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decin
default	delegate	do	double	else	enum	even
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	parar
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struc
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					
上下文关键字						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partia (type
partial (method)	remove	select	set			

C# 数据类型

在 C# 中，变量分为以下几种类型：

- 值类型（Value types）
- 引用类型（Reference types）
- 指针类型（Pointer types）

值类型（Value types）

值类型变量可以直接分配给一个值。它们是从类 **System.ValueType** 中派生的。

值类型直接包含数据。比如 **int**、**char**、**float**，它们分别存储数字、字母、浮点数。当您声明一个 **int** 类型时，系统分配内存来存储值。

下表列出了 C# 2010 中可用的值类型：

类型	描述	范围	默认值
bool	布尔值	True 或 False	False
byte	8 位无符号整数	0 到 255	0
char	16 位 Unicode 字符	U +0000 到 U +ffff	'\0'
decimal	128 位精确的十进制值，28-29 有效位数	$(-7.9 \times 10^{28}$ 到 $7.9 \times 10^{28}) / 10^{0}$ 到 28	0.0M
double	64 位双精度浮点型	$(+/-)5.0 \times 10^{-324}$ 到 $(+/-)1.7 \times 10^{308}$	0.0D
float	32 位单精度浮点型	-3.4×10^{38} 到 $+ 3.4 \times 10^{38}$	0.0F
int	32 位有符号整数类型	-2,147,483,648 到 2,147,483,647	0
long	64 位有符号整数类型	-923,372,036,854,775,808 到 9,223,372,036,854,775,807	0L
sbyte	8 位有符号整数类型	-128 到 127	0
short	16 位有符号整数类型	-32,768 到 32,767	0
uint	32 位无符号整数类型	0 到 4,294,967,295	0
ulong	64 位无符号整数类型	0 到 18,446,744,073,709,551,615	0
ushort	16 位无符号整数类型	0 到 65,535	0

如需得到一个类型或一个变量在特定平台上的准确尺寸，可以使用 **sizeof** 方法。表达式 *sizeof(type)* 产生以字节为单位存储对象或类型的存储尺寸。下面举例获取任何机器上 *int* 类型的存储尺寸：

```
namespace DataTypeApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Size of int: {0}", sizeof(int));
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Size of int: 4
```

引用类型（Reference types）

引用类型不包含存储在变量中的实际数据，但它们包含对变量的引用。

换句话说，它们指的是一个内存位置。使用多个变量时，引用类型可以指向一个内存位置。如果内存位置的数据是由一个变量改变的，其他变量会自动反映这种值的变化。内置的 引用类型有：**object**、**dynamic** 和 **string**。

对象（Object）类型

对象（**Object**）类型 是 C# 通用类型系统（Common Type System - CTS）中所有数据类型的终极基类。Object 是 System.Object 类的别名。所以对象（Object）类型可以被分配任何其他类型（值类型、引用类型、预定义类型或用户自定义类型）的值。但是，在分配值之前，需要先进行类型转换。

当一个值类型转换为对象类型时，则被称为 装箱；另一方面，当一个对象类型转换为值类型时，则被称为 拆箱。

```
object obj;
obj = 100; // 这是装箱
```

动态（Dynamic）类型

您可以存储任何类型的值在动态数据类型变量中。这些变量的类型检查是在运行时发生的。

声明动态类型的语法：

```
dynamic <variable_name> = value;
```

例如：

```
dynamic d = 20;
```

动态类型与对象类型相似，但是对象类型变量的类型检查是在编译时发生的，而动态类型变量的类型检查是在运行时发生的。

字符串（String）类型

字符串（**String**）类型 允许您给变量分配任何字符串值。字符串（String）类型是 System.String 类的别名。它是从对象（Object）类型派生的。字符串（String）类型的值可以通过两种形式进行分配：引号和 @引号。

例如：

```
String str = "w3cschool.cc";
```

一个 @引号字符串：

```
@"w3cschool.cc";
```

C# string 字符串的前面可以加 @（称作"逐字字符串"）将转义字符（\）当作普通字符对待，比如：

```
string str = @"C:\Windows";
```

等价于：

```
string str = "C:\\Windows";
```

@ 字符串中可以任意换行，换行符及缩进空格都计算在字符串长度之内。

```
string str = @"<script type=""text/javascript"">
    <!--
    -->
    </script>";
```

用户自定义引用类型有：class、interface 或 delegate。我们将在以后的章节中讨论这些类型。

指针类型（Pointer types）

指针类型变量存储另一种类型的内存地址。C# 中的指针与 C 或 C++ 中的指针有相同的功能。

声明指针类型的语法：

```
type* identifier;
```

例如：

```
char* cptr;
int* iptr;
```

我们将在章节"不安全的代码"中讨论指针类型。

C# 类型转换

类型转换从根本上说是类型铸造，或者说是把数据从一种类型转换为另一种类型。在 C# 中，类型铸造有两种形式：

- 隐式类型转换 - 这些转换是 C# 默认的以安全方式进行的转换。例如，从小的整数类型转换为大的整数类型，从派生类转换为基类。
- 显式类型转换 - 这些转换是通过用户使用预定义的函数显式完成的。显式转换需要强制转换运算符。

下面的实例显示了一个显式的类型转换：

```
namespace TypeConversionApplication
{
    class ExplicitConversion
    {
        static void Main(string[] args)
        {
            double d = 5673.74;
            int i;

            // 强制转换 double 为 int
            i = (int)d;
            Console.WriteLine(i);
            Console.ReadKey();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
5673
```

C# 类型转换方法

C# 提供了下列内置的类型转换方法：

方法	描述
ToBoolean	如果可能的话，把类型转换为布尔型。
ToByte	把类型转换为字节类型。
ToChar	如果可能的话，把类型转换为单个 Unicode 字符类型。
ToDateTime	把类型（整数或字符串类型）转换为 日期-时间 结构。
ToDecimal	把浮点型或整数类型转换为十进制类型。
ToDouble	把类型转换为双精度浮点型。
ToInt16	把类型转换为 16 位整数类型。
ToInt32	把类型转换为 32 位整数类型。
ToInt64	把类型转换为 64 位整数类型。
ToSbyte	把类型转换为有符号字节类型。
ToSingle	把类型转换为小浮点数类型。
ToString	把类型转换为字符串类型。
ToType	把类型转换为指定类型。
ToUInt16	把类型转换为 16 位无符号整数类型。
ToUInt32	把类型转换为 32 位无符号整数类型。
ToUInt64	把类型转换为 64 位无符号整数类型。

下面的实例把不同值的类型转换为字符串类型：

```
namespace TypeConversionApplication
{
    class StringConversion
    {
        static void Main(string[] args)
        {
            int i = 75;
            float f = 53.005f;
            double d = 2345.7652;
            bool b = true;

            Console.WriteLine(i.ToString());
            Console.WriteLine(f.ToString());
            Console.WriteLine(d.ToString());
            Console.WriteLine(b.ToString());
            Console.ReadKey();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
75
53.005
2345.7652
True
```

C# 变量

一个变量只不过是一个供程序操作的存储区的名字。在 C# 中，每个变量都有一个特定的类型，类型决定了变量的内存大小和布局。范围内的值可以存储在内存中，可以对变量进行一系列操作。

我们已经讨论了各种数据类型。C# 中提供的基本的值类型大致可以分为以下几类：

类型	举例
整数类型	sbyte、byte、short、ushort、int、uint、long、ulong 和 char
浮点型	float 和 double
十进制类型	decimal
布尔类型	true 或 false 值，指定的值
空类型	可为空值的数据类型

C# 允许定义其他值类型的变量，比如 **enum**，也允许定义引用类型变量，比如 **class**。这些我们将在以后的章节中进行讨论。在本章节中，我们只研究基本变量类型。

C# 中的变量定义

C# 中变量定义的语法：

```
<data_type> <variable_list>;
```

在这里，data_type 必须是一个有效的 C# 数据类型，可以是 char、int、float、double 或其他用户自定义的数据类型。variable_list 可以由一个或多个用逗号分隔的标识符名称组成。

一些有效的变量定义如下所示：

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

您可以在变量定义时进行初始化：

```
int i = 100;
```

C# 中的变量初始化

变量通过在等号后跟一个常量表达式进行初始化（赋值）。初始化的一般形式为：

```
variable_name = value;
```

变量可以在声明时被初始化（指定一个初始值）。初始化由一个等号后跟一个常量表达式组成，如下所示：

```
<data_type> <variable_name> = value;
```

一些实例：

```
int d = 3, f = 5;    /* 初始化 d 和 f. */
byte z = 22;         /* 初始化 z. */
double pi = 3.14159; /* 声明 pi 的近似值 */
char x = 'x';        /* 变量 x 的值为 'x' */
```

正确地初始化变量是一个良好的编程习惯，否则有时程序会产生意想不到的结果。

请看下面的实例，使用了各种类型的变量：

```
namespace VariableDefinition
{
    class Program
    {
        static void Main(string[] args)
        {
            short a;
            int b ;
            double c;

            /* 实际初始化 */
            a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a = 10, b = 20, c = 30
```

接受来自用户的值

System 命名空间中的 **Console** 类提供了一个函数 **ReadLine()**，用于接收来自用户的输入，并把它存储到一个变量中。

例如：

```
int num;
num = Convert.ToInt32(Console.ReadLine());
```

函数 **Convert.ToInt32()** 把用户输入的数据转换为 int 数据类型，因为 **Console.ReadLine()** 只接受字符串格式的数据。

C# 中的 Lvalues 和 Rvalues

C# 中的两种表达式：

1. **lvalue**：lvalue 表达式可以出现在赋值语句的左边或右边。
2. **rvalue**：rvalue 表达式可以出现在赋值语句的右边，不能出现在赋值语句的左边。

变量是 lvalue 的，所以可以出现在赋值语句的左边。数值是 rvalue 的，因此不能被赋值，不能出现在赋值语句的左边。下面是一个有效的语句：

```
int g = 20;
```

下面是一个无效的语句，会产生编译时错误：

```
10 = 20;
```

C# 常量

常量是固定值，程序执行期间不会改变。常量可以是任何基本数据类型，比如整数常量、浮点常量、字符常量或者字符串常量，还有枚举常量。

常量可以被当作常规的变量，只是它们的值在定义后不能被修改。

整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，没有前缀则表示十进制。

整数常量也可以有后缀，可以是 U 和 L 的组合，其中，U 和 L 分别表示 unsigned 和 long。后缀可以是大写或者小写，多个后缀以任意顺序进行组合。

这里有一些整数常量的实例：

```
212         /* 合法 */
215u        /* 合法 */
0xFeeL      /* 合法 */
078         /* 非法：8 不是一个八进制数字 */
032UU       /* 非法：不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85         /* 十进制 */
0213       /* 八进制 */
0x4b       /* 十六进制 */
30         /* int */
30u        /* 无符号 int */
30l        /* long */
30ul       /* 无符号 long */
```

浮点常量

一个浮点常量是由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

这里有一些浮点常量的实例：

```
3.14159       /* 合法 */
314159E-5L    /* 合法 */
510E          /* 非法：不完全指数 */
210f          /* 非法：没有小数或指数 */
.e55          /* 非法：缺少整数或小数 */
```

使用小数形式表示时，必须包含小数点、指数或同时包含两者。使用指数形式表示时，必须包含整数部分、小数部分或同时包含两者。有符号的指数是用 e 或 E 表示的。

字符常量

字符常量是括在单引号里，例如，'x'，且可存储在一个简单的字符类型变量中。一个字符常量可以是一个普通字符（例如 'x'）、一个转义序列（例如 '\t'）或者一个通用字符（例如 '\u02C0'）。

在 C# 中有一些特定的字符，当它们的前面带有反斜杠时有特殊的意义，可用于表示换行符（\n）或制表符 tab（\t）。在这里，列出一些转义序列码：

转义序列	含义
\	\ 字符
\'	' 字符
\"	" 字符
\?	? 字符
\a	Alert 或 bell
\b	退格键（Backspace）
\f	换页符（Form feed）
\n	换行符（Newline）
\r	回车
\t	水平制表符 tab
\v	垂直制表符 tab
\ooo	一到三位的八进制数
\xhh . . .	一个或多个数字的十六进制数

以下是一些转义序列字符的实例：

```
namespace EscapeChar
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello\tWorld\n\n");
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello   World
```

字符串常量

字符常量是括在双引号 "" 里，或者是括在 @"" 里。字符串常量包含的字符与字符常量相似，可以是：普通字符、转义序列和通用字符

使用字符串常量时，可以把一个很长的行拆成多个行，可以使用空格分隔各个部分。

这里是一些字符串常量的实例。下面所列的各种形式表示相同的字符串。

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

定义常量

常量是使用 **const** 关键字来定义的 。定义一个常量的语法如下：

```
const <data_type> <constant_name> = value;
```

下面的代码演示了如何在程序中定义和使用常量：

```
using System;

namespace DeclaringConstants
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159; // 常量声明
            double r;
            Console.WriteLine("Enter Radius: ");
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Enter Radius:
3
Radius: 3, Area: 28.27431
```

C# 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C# 有丰富的内置运算符，分类如下：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

本教程将逐一讲解算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符及其他运算符。

算术运算符

下表显示了 C# 支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

实例

请看下面的实例，了解 C# 中所有可用的算术运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int b = 10;
            int c;

            c = a + b;
            Console.WriteLine("Line 1 - c 的值是 {0}", c);
            c = a - b;
            Console.WriteLine("Line 2 - c 的值是 {0}", c);
            c = a * b;
            Console.WriteLine("Line 3 - c 的值是 {0}", c);
            c = a / b;
            Console.WriteLine("Line 4 - c 的值是 {0}", c);
            c = a % b;
            Console.WriteLine("Line 5 - c 的值是 {0}", c);
            c = a++;
            Console.WriteLine("Line 6 - c 的值是 {0}", c);
            c = a--;
            Console.WriteLine("Line 7 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 21
Line 7 - c 的值是 22
```

关系运算符

下表显示了 C# 支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

实例

请看下面的实例，了解 C# 中所有可用的关系运算符：

```
using System;

class Program
{
  static void Main(string[] args)
  {
      int a = 21;
      int b = 10;

      if (a == b)
      {
          Console.WriteLine("Line 1 - a 等于 b");
      }
      else
      {
          Console.WriteLine("Line 1 - a 不等于 b");
      }
      if (a < b)
      {
          Console.WriteLine("Line 2 - a 小于 b");
      }
      else
      {
          Console.WriteLine("Line 2 - a 不小于 b");
      }
      if (a > b)
      {
          Console.WriteLine("Line 3 - a 大于 b");
      }
      else
      {
          Console.WriteLine("Line 3 - a 不大于 b");
      }
      /* 改变 a 和 b 的值 */
      a = 5;
      b = 20;
      if (a <= b)
      {
         Console.WriteLine("Line 4 - a 小于或等于 b");
      }
      if (b >= a)
      {
         Console.WriteLine("Line 5 - b 大于或等于 a");
      }
  }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - a 不等于 b
Line 2 - a 不小于 b
Line 3 - a 大于 b
Line 4 - a 小于或等于 b
Line 5 - b 大于或等于 a
```

逻辑运算符

下表显示了 C# 支持的所有逻辑运算符。假设变量 **A** 为布尔值 true，变量 **B** 为布尔值 false，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
|:|:	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A |:|: B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

实例

请看下面的实例，了解 C# 中所有可用的逻辑运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            bool a = true;
            bool b = true;

            if (a && b)
            {
               Console.WriteLine("Line 1 - 条件为真");
            }
            if (a || b)
            {
                Console.WriteLine("Line 2 - 条件为真");
            }
            /* 改变 a 和 b 的值 */
            a = false;
            b = true;
            if (a && b)
            {
                Console.WriteLine("Line 3 - 条件为真");
            }
            else
            {
                Console.WriteLine("Line 3 - 条件不为真");
            }
            if (!(a && b))
            {
                Console.WriteLine("Line 4 - 条件为真");
            }
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

位运算符

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

下表列出了 C# 支持的位运算符。假设变量 **A** 的值为 60，变量 **B** 的值为 13，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果。	(~A) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

请看下面的实例，了解 C# 中所有可用的位运算符：

```
using System;
namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 60;               /* 60 = 0011 1100 */
            int b = 13;               /* 13 = 0000 1101 */
            int c = 0;

             c = a & b;           /* 12 = 0000 1100 */
             Console.WriteLine("Line 1 - c 的值是 {0}", c );

             c = a | b;           /* 61 = 0011 1101 */
             Console.WriteLine("Line 2 - c 的值是 {0}", c);

             c = a ^ b;           /* 49 = 0011 0001 */
             Console.WriteLine("Line 3 - c 的值是 {0}", c);

             c = ~a;               /*-61 = 1100 0011 */
             Console.WriteLine("Line 4 - c 的值是 {0}", c);

             c = a << 2;     /* 240 = 1111 0000 */
             Console.WriteLine("Line 5 - c 的值是 {0}", c);

             c = a >> 2;     /* 15 = 0000 1111 */
             Console.WriteLine("Line 6 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

赋值运算符

下表列出了 C# 支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C = *A* 相当于 *C* = *C* A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
|:=	按位或且赋值运算符	C |:= 2 等同于 C = C |: 2

实例

请看下面的实例，了解 C# 中所有可用的赋值运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int c;

            c = a;
            Console.WriteLine("Line 1 - =  c 的值 = {0}", c);

            c += a;
            Console.WriteLine("Line 2 - += c 的值 = {0}", c);

            c -= a;
            Console.WriteLine("Line 3 - -=  c 的值 = {0}", c);

            c *= a;
            Console.WriteLine("Line 4 - *=  c 的值 = {0}", c);

            c /= a;
            Console.WriteLine("Line 5 - /=  c 的值 = {0}", c);

            c = 200;
            c %= a;
            Console.WriteLine("Line 6 - %=  c 的值 = {0}", c);

            c <<= 2;
            Console.WriteLine("Line 7 - <<=  c 的值 = {0}", c);

            c >>= 2;
            Console.WriteLine("Line 8 - >>=  c 的值 = {0}", c);

            c &= 2;
            Console.WriteLine("Line 9 - &=  c 的值 = {0}", c);

            c ^= 2;
            Console.WriteLine("Line 10 - ^=  c 的值 = {0}", c);

            c |= 2;
            Console.WriteLine("Line 11 - |=  c 的值 = {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - =     c 的值 = 21
Line 2 - +=    c 的值 = 42
Line 3 - -=    c 的值 = 21
Line 4 - *=    c 的值 = 441
Line 5 - /=    c 的值 = 21
Line 6 - %=    c 的值 = 11
Line 7 - <<=    c 的值 = 44
Line 8 - >>=    c 的值 = 11
Line 9 - &=    c 的值 = 2
Line 10 - ^=    c 的值 = 0
Line 11 - |=    c 的值 = 2
```

杂项运算符

下表列出了 C# 支持的其他一些重要的运算符，包括 **sizeof**、**typeof** 和 **? :**。

运算符	描述	实例
sizeof()	返回数据类型的大小。	sizeof(int)，将返回 4.
typeof()	返回 class 的类型。	typeof(StreamReader);
&	返回变量的地址。	&a; 将得到变量的实际地址。
*	变量的指针。	*a; 将指向一个变量。
? :	条件表达式	如果条件为真 ? 则为 X : 否则为 Y
is	判断对象是否为某一类型。	If(Ford is Car) // 检查 Ford 是否是 Car 类的一个对象。
as	强制转换，即使转换失败也不会抛出异常。	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

实例

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {

         /* sizeof 运算符的实例 */
         Console.WriteLine("int 的大小是 {0}", sizeof(int));
         Console.WriteLine("short 的大小是 {0}", sizeof(short));
         Console.WriteLine("double 的大小是 {0}", sizeof(double));

         /* 三元运算符符的实例 */
         int a, b;
         a = 10;
         b = (a == 1) ? 20 : 30;
         Console.WriteLine("b 的值是 {0}", b);

         b = (a == 10) ? 20 : 30;
         Console.WriteLine("b 的值是 {0}", b);
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
int 的大小是 4
short 的大小是 2
double 的大小是 8
b 的值是 30
b 的值是 20
```

C# 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 2，在这里，*x* 被赋值为 *13*，而不是 *20*，因为运算符 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	() [] -> . ++ - -	从左到右
一元	+ - ! ~ ++ - - (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR			从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

实例

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {
         int a = 20;
         int b = 10;
         int c = 15;
         int d = 5;
         int e;
         e = (a + b) * c / d;     // ( 30 * 15 ) / 5
         Console.WriteLine("(a + b) * c / d 的值是 {0}", e);

         e = ((a + b) * c) / d;   // (30 * 15 ) / 5
         Console.WriteLine("((a + b) * c) / d 的值是 {0}", e);

         e = (a + b) * (c / d);   // (30) * (15/5)
         Console.WriteLine("(a + b) * (c / d) 的值是 {0}", e);

         e = a + (b * c) / d;    //  20 + (150/5)
         Console.WriteLine("a + (b * c) / d 的值是 {0}", e);
         Console.ReadLine();
      }
   }
}
```

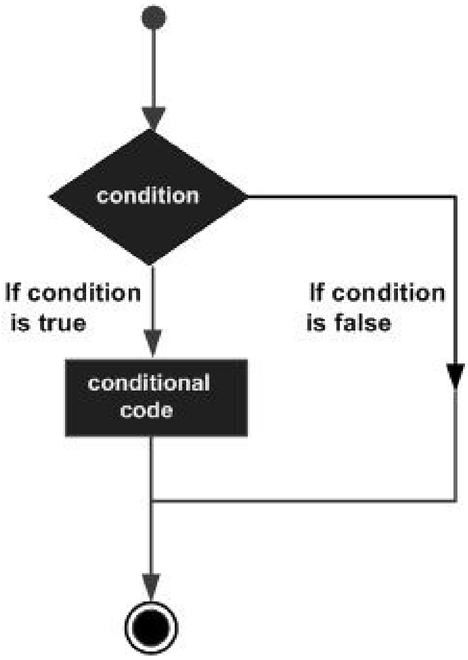
当上面的代码被编译和执行时，它会产生下列结果：

```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

C# 判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

下面是大多数编程语言中典型的判断结构的一般形式：



判断语句

C# 提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
if 语句	一个 **if 语句** 由一个布尔表达式后跟一个或多个语句组成。
if...else 语句	一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。
嵌套 if 语句	您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。
switch 语句	一个 **switch** 语句允许测试一个变量等于多个值时的情况。
嵌套 switch 语句	您可以在一个 **switch** 语句内使用另一个 **switch** 语句。

? : 运算符

我们已经在前面的章节中讲解了 条件运算符 **? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

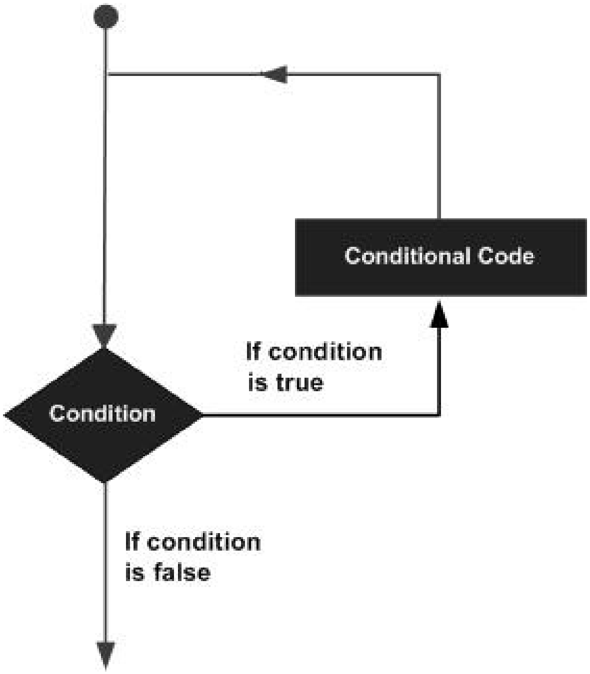
? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

C# 循环

有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：



循环类型

C# 提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 while、for 或 do..while 循环内使用一个或多个循环。

循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C# 提供了下列的控制语句。点击链接查看每个语句的细节。

控制语句	描述
break 语句	终止 **loop** 或 **switch** 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。
continue 语句	引起循环跳过主体的剩余部分，立即重新开始测试条件。

无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            for (; ; )
            {
                Console.WriteLine("Hey! I am Trapped");
            }

        }
    }
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，程序员偏向于使用 for(;;) 结构来表示一个无限循环。

C# 封装

封装 被定义为"把一个或多个项目封闭在一个物理的或者逻辑的包中"。在面向对象程序设计方法论中，封装是为了防止对实现细节的访问。

抽象和封装是面向对象程序设计的相关特性。抽象允许相关信息可视化，封装则使程序员实现所需级别的抽象。

封装使用 访问修饰符 来实现。一个 访问修饰符 定义了一个类成员的范围和可见性。C# 支持的访问修饰符如下所示：

- Public
- Private
- Protected
- Internal
- Protected internal

Public 访问修饰符

Public 访问修饰符允许一个类将其成员变量和成员函数暴露给其他的函数和对象。任何公有成员可以被外部的类访问。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //成员变量
        public double length;
        public double width;

        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("长度： {0}", length);
            Console.WriteLine("宽度： {0}", width);
            Console.WriteLine("面积： {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
长度： 4.5
宽度： 3.5
面积： 15.75
```

在上面的实例中，成员变量 length 和 width 被声明为 **public**，所以它们可以被函数 Main() 使用 Rectangle 类的实例 **r** 访问。

成员函数 *Display()* 和 *GetArea()* 也可以不通过类的实例直接访问这些变量。

成员函数 *Display()* 也被声明为 **public**，所以它也能被 *Main()* 使用 Rectangle 类的实例 **r** 访问。

Private 访问修饰符

Private 访问修饰符允许一个类将其成员变量和成员函数对其他的函数和对象进行隐藏。只有同一个类中的函数可以访问它的私有成员。即使是类的实例也不能访问它的私有成员。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //成员变量
        private double length;
        private double width;

        public void Acceptdetails()
        {
            Console.WriteLine("请输入长度：");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("请输入宽度：");
            width = Convert.ToDouble(Console.ReadLine());
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("长度： {0}", length);
            Console.WriteLine("宽度： {0}", width);
            Console.WriteLine("面积： {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
请输入长度：
4.4
请输入宽度：
3.3
长度： 4.4
宽度： 3.3
面积： 14.52
```

在上面的实例中，成员变量 length 和 width 被声明为 **private**，所以它们不能被函数 Main() 访问。

成员函数 *AcceptDetails()* 和 *Display()* 可以访问这些变量。

由于成员函数 *AcceptDetails()* 和 *Display()* 被声明为 **public**，所以它们可以被 *Main()* 使用 Rectangle 类的实例 **r** 访问。

Protected 访问修饰符

Protected 访问修饰符允许子类访问它的基类的成员变量和成员函数。这样有助于实现继承。我们将在继承的章节详细讨论这个。更详细地讨论这个。

Internal 访问修饰符

Internal 访问说明符允许一个类将其成员变量和成员函数暴露给当前程序中的其他函数和对象。换句话说，带有 internal 访问修饰符的任何成员可以被定义在该成员所定义的应用程序内的任何类或方法访问。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //成员变量
        internal double length;
        internal double width;

        double GetArea()
        {
            return length * width;
        }
       public void Display()
        {
            Console.WriteLine("长度： {0}", length);
            Console.WriteLine("宽度： {0}", width);
            Console.WriteLine("面积： {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
长度： 4.5
宽度： 3.5
面积： 15.75
```

在上面的实例中，请注意成员函数 *GetArea()* 声明的时候不带有任何访问修饰符。如果没有指定访问修饰符，则使用类成员的默认访问修饰符，即为 **private**。

Protected Internal 访问修饰符

Protected Internal 访问修饰符允许一个类将其成员变量和成员函数对同一应用程序内的子类以外的其他的类对象和函数进行隐藏。这也被用于实现继承。

C# 方法

一个方法是把一些相关的语句组织在一起，用来执行一个任务的语句块。每一个 C# 程序至少有一个带有 Main 方法的类。

要使用一个方法，您需要：

- 定义方法
- 调用方法

C# 中定义方法

当定义一个方法时，从根本上说是在声明它的结构的元素。在 C# 中，定义方法的语法如下：

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

下面是方法的各个元素：

- **Access Specifier**：访问修饰符，这个决定了变量或方法对于另一个类的可见性。
- **Return type**：返回类型，一个方法可以返回一个值。返回类型是方法返回的值的数据类型。如果方法不返回任何值，则返回类型为 **void**。
- **Method name**：方法名称，是一个唯一的标识符，且是大小写敏感的。它不能与类中声明的其他标识符相同。
- **Parameter list**：参数列表，使用圆括号括起来，该参数是用来传递和接收方法的数据。参数列表是指方法的参数类型、顺序和数量。参数是可选的，也就是说，一个方法可能不包含参数。
- **Method body**：方法主体，包含了完成任务所需的指令集。

实例

下面的代码片段显示一个函数 *FindMax*，它接受两个整数值，并返回两个中的较大值。它有 public 访问修饰符，所以它可以使用类的实例从类的外部进行访问。

```
class NumberManipulator
{
   public int FindMax(int num1, int num2)
   {
      /* 局部变量声明 */
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

C# 中调用方法

您可以使用方法名调用方法。下面的实例演示了这点：

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public int FindMax(int num1, int num2)
      {
         /* 局部变量声明 */
         int result;

         if (num1 > num2)
            result = num1;
         else
            result = num2;

         return result;
      }
      static void Main(string[] args)
      {
         /* 局部变量定义 */
         int a = 100;
         int b = 200;
         int ret;
         NumberManipulator n = new NumberManipulator();

         //调用 FindMax 方法
         ret = n.FindMax(a, b);
         Console.WriteLine("最大值是： {0}", ret );
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
最大值是： 200
```

您也可以使用类的实例从另一个类中调用其他类的公有方法。例如，方法 *FindMax* 属于 *NumberManipulator* 类，您可以从另一个类 *Test* 中调用它。

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            /* 局部变量声明 */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            /* 局部变量定义 */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();
            //调用 FindMax 方法
            ret = n.FindMax(a, b);
            Console.WriteLine("最大值是： {0}", ret );
            Console.ReadLine();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
最大值是： 200
```

递归方法调用

一个方法可以自我调用。这就是所谓的 递归。下面的实例使用递归函数计算一个数的阶乘：

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int factorial(int num)
        {
            /* 局部变量定义 */
            int result;

            if (num == 1)
            {
                return 1;
            }
            else
            {
                result = factorial(num - 1) * num;
                return result;
            }
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            //调用 factorial 方法
            Console.WriteLine("6 的阶乘是： {0}", n.factorial(6));
            Console.WriteLine("7 的阶乘是： {0}", n.factorial(7));
            Console.WriteLine("8 的阶乘是： {0}", n.factorial(8));
            Console.ReadLine();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
6 的阶乘是： 720
7 的阶乘是： 5040
8 的阶乘是： 40320
```

参数传递

当调用带有参数的方法时，您需要向方法传递参数。在 C# 中，有三种向方法传递参数的方式：

方式	描述
值参数	这种方式复制参数的实际值给函数的形式参数，实参和形参使用的是两个不同内存中的值。在这种情况下，当形参的值发生改变时，不会影响实参的值，从而保证了实参数据的安全。
引用参数	这种方式复制参数的内存位置的引用给形式参数。这意味着，当形参的值发生改变时，同时也改变实参的值。
输出参数	这种方式可以返回多个值。

按值传递参数

这是参数传递的默认方式。在这种方式下，当调用一个方法时，会为每个值参数创建一个新的存储位置。

实际参数的值会复制给形参，实参和形参使用的是两个不同内存中的值。所以，当形参的值发生改变时，不会影响实参的值，从而保证了实参数据的安全。下面的实例演示了这个概念：

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public void swap(int x, int y)
        {
            int temp;

            temp = x; /* 保存 x 的值 */
            x = y;    /* 把 y 赋值给 x */
            y = temp; /* 把 temp 赋值给 y */
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            /* 局部变量定义 */
            int a = 100;
            int b = 200;

            Console.WriteLine("在交换之前，a 的值： {0}", a);
            Console.WriteLine("在交换之前，b 的值： {0}", b);

            /* 调用函数来交换值 */
            n.swap(a, b);

            Console.WriteLine("在交换之后，a 的值： {0}", a);
            Console.WriteLine("在交换之后，b 的值： {0}", b);

            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在交换之前，a 的值：100
在交换之前，b 的值：200
在交换之后，a 的值：100
在交换之后，b 的值：200
```

结果表明，即使在函数内改变了值，值也没有发生任何的变化。

按引用传递参数

引用参数是一个对变量的内存位置的引用。当按引用传递参数时，与值参数不同的是，它不会为这些参数创建一个新的存储位置。引用参数表示与提供给方法的实际参数具有相同的内存位置。

在 C# 中，使用 **ref** 关键字声明引用参数。下面的实例演示了这点：

```
using System;
namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void swap(ref int x, ref int y)
      {
         int temp;

         temp = x; /* 保存 x 的值 */
         x = y;    /* 把 y 赋值给 x */
         y = temp; /* 把 temp 赋值给 y */
       }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* 局部变量定义 */
         int a = 100;
         int b = 200;

         Console.WriteLine("在交换之前，a 的值： {0}", a);
         Console.WriteLine("在交换之前，b 的值： {0}", b);

         /* 调用函数来交换值 */
         n.swap(ref a, ref b);

         Console.WriteLine("在交换之后，a 的值： {0}", a);
         Console.WriteLine("在交换之后，b 的值： {0}", b);

         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在交换之前，a 的值：100
在交换之前，b 的值：200
在交换之后，a 的值：200
在交换之后，b 的值：100
```

结果表明，*swap* 函数内的值改变了，且这个改变可以在 *Main* 函数中反映出来。

按输出传递参数

return 语句可用于只从函数中返回一个值。但是，可以使用 输出参数 来从函数中返回两个值。输出参数会把方法输出的数据赋给自己，其他方面与引用参数相似。

下面的实例演示了这点：

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValue(out int x )
      {
         int temp = 5;
         x = temp;
      }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* 局部变量定义 */
         int a = 100;

         Console.WriteLine("在方法调用之前，a 的值： {0}", a);

         /* 调用函数来获取值 */
         n.getValue(out a);

         Console.WriteLine("在方法调用之后，a 的值： {0}", a);
         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在方法调用之前，a 的值： 100
在方法调用之后，a 的值： 5
```

提供给输出参数的变量不需要赋值。当需要从一个参数没有指定初始值的方法中返回值时，输出参数特别有用。请看下面的实例，来理解这一点：

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValues(out int x, out int y )
      {
          Console.WriteLine("请输入第一个值： ");
          x = Convert.ToInt32(Console.ReadLine());
          Console.WriteLine("请输入第二个值： ");
          y = Convert.ToInt32(Console.ReadLine());
      }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* 局部变量定义 */
         int a , b;

         /* 调用函数来获取值 */
         n.getValues(out a, out b);

         Console.WriteLine("在方法调用之后，a 的值： {0}", a);
         Console.WriteLine("在方法调用之后，b 的值： {0}", b);
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果（取决于用户输入）：

```
请输入第一个值：
7
请输入第二个值：
8
在方法调用之后，a 的值： 7
在方法调用之后，b 的值： 8
```

C# 可空类型（Nullable）

C# 可空类型（Nullable）

C# 提供了一个特殊的数据类型，**nullable** 类型（可空类型），可空类型可以表示其基础值类型正常范围内的值，再加上一个 null 值。

例如，Nullable< Int32 >，读作"可空的 Int32"，可以被赋值为 -2,147,483,648 到 2,147,483,647 之间的任意值，也可以被赋值为 null 值。类似的，Nullable< bool > 变量可以被赋值为 true 或 false 或 null。

在处理数据库和其他包含可能未赋值的元素的数据类型时，将 null 赋值给数值类型或布尔型的功能特别有用。例如，数据库中的布尔型字段可以存储值 true 或 false，或者，该字段也可以未定义。

声明一个 **nullable** 类型（可空类型）的语法如下：

```
< data_type> ? <variable_name> = null;
```

下面的实例演示了可空数据类型的用法：

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {
      static void Main(string[] args)
      {
         int? num1 = null;
         int? num2 = 45;
         double? num3 = new double?();
         double? num4 = 3.14157;

         bool? boolval = new bool?();

         // 显示值

         Console.WriteLine("显示可空类型的值： {0}, {1}, {2}, {3}",
                            num1, num2, num3, num4);
         Console.WriteLine("一个可空的布尔值： {0}", boolval);
         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
显示可空类型的值： , 45,  , 3.14157
一个可空的布尔值：
```

Null 合并运算符（ ?? ）

Null 合并运算符用于定义可空类型和引用类型的默认值。Null 合并运算符为类型转换定义了一个预设值，以防可空类型的值为 Null。Null 合并运算符把操作数类型隐式转换为另一个可空（或不可空）的值类型的操作数的类型。

如果第一个操作数的值为 null，则运算符返回第二个操作数的值，否则返回第一个操作数的值。下面的实例演示了这点：

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {

      static void Main(string[] args)
      {

         double? num1 = null;
         double? num2 = 3.14157;
         double num3;
         num3 = num1 ?? 5.34;
         Console.WriteLine("num3 的值： {0}", num3);
         num3 = num2 ?? 5.34;
         Console.WriteLine("num3 的值： {0}", num3);
         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

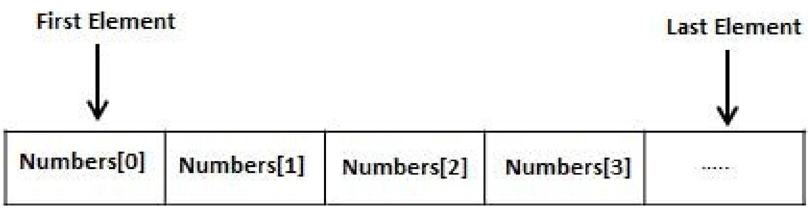
```
num3 的值： 5.34
num3 的值： 3.14157
```

C# 数组（Array）

数组是一个存储相同类型元素的固定大小的顺序集合。数组是用来存储数据的集合，通常认为数组是一个同一类型变量的集合。

声明数组变量并不是声明 number0、number1、...、number99 一个个单独的变量，而是声明一个就像 numbers 这样的变量，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

所有的数组都是由连续的内存位置组成的。最低的地址对应第一个元素，最高的地址对应最后一个元素。



声明数组

在 C# 中声明一个数组，您可以使用下面的语法：

```
datatype[] arrayName;
```

其中，

- *datatype* 用于指定被存储在数组中的元素的类型。
- *[ ]* 指定数组的秩（维度）。秩指定数组的大小。
- *arrayName* 指定数组的名称。

例如：

```
double[] balance;
```

初始化数组

声明一个数组不会在内存中初始化数组。当初始化数组变量时，您可以赋值给数组。

数组是一个引用类型，所以您需要使用 **new** 关键字来创建数组的实例。

例如：

```
double[] balance = new double[10];
```

赋值给数组

您可以通过使用索引号赋值给一个单独的数组元素，比如：

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

您可以在声明数组的同时给数组赋值，比如：

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

您也可以创建并初始化一个数组，比如：

```
int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

在上述情况下，你也可以省略数组的大小，比如：

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

您也可以赋值一个数组变量到另一个目标数组变量中。在这种情况下，目标和源会指向相同的内存位置：

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
int[] score = marks;
```

当您创建一个数组时，C# 编译器会根据数组类型隐式初始化每个数组元素为一个默认值。例如，int 数组的所有元素都会被初始化为 0。

访问数组元素

元素是通过带索引的数组名称来访问的。这是通过把元素的索引放置在数组名称后的方括号中来实现的。例如：

```
double salary = balance[9];
```

下面是一个实例，使用上面提到的三个概念，即声明、赋值、访问数组：

```
using System;
namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n 是一个带有 10 个整数的数组 */
         int i,j;

         /* 初始化数组 n 中的元素 */
         for ( i = 0; i < 10; i++ )
         {
            n[ i ] = i + 100;
         }

         /* 输出每个数组元素的值 */
         for (j = 0; j < 10; j++ )
         {
            Console.WriteLine("Element[{0}] = {1}", j, n[j]);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

使用 *foreach* 循环

在前面的实例中，我们使用一个 for 循环来访问每个数组元素。您也可以使用一个 **foreach** 语句来遍历数组。

```
using System;

namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n 是一个带有 10 个整数的数组 */

         /* 初始化数组 n 中的元素 */
         for ( int i = 0; i < 10; i++ )
         {
            n[i] = i + 100;
         }

         /* 输出每个数组元素的值 */
         foreach (int j in n )
         {
            int i = j-100;
            Console.WriteLine("Element[{0}] = {1}", i, j);
            i++;
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C# 数组细节

在 C# 中，数组是非常重要的，且需要了解更多的细节。下面列出了 C# 程序员必须清楚的一些与数组相关的重要概念：

概念	描述
多维数组	C# 支持多维数组。多维数组最简单的形式是二维数组。
交错数组	C# 支持交错数组，即数组的数组。
传递数组给函数	您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
参数数组	这通常用于传递未知数量的参数给函数。
Array 类	在 System 命名空间中定义，是所有数组的基类，并提供了各种用于数组的属性和方法。

C# 字符串（String）

在 C# 中，您可以使用字符数组来表示字符串，但是，更常见的做法是使用 **string** 关键字来声明一个字符串变量。string 关键字是 **System.String** 类的别名。

创建 String 对象

您可以使用以下方法之一来穿件 string 对象：

- 通过给 String 变量指定一个字符串
- 通过使用 String 类构造函数
- 通过使用字符串串联运算符（ + ）
- 通过检索属性或调用一个返回字符串的方法
- 通过格式化方法来转换一个值或对象为它的字符串表示形式

下面的实例演示了这点：

```
using System;

namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           //字符串，字符串连接
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //通过使用 string 构造函数
            char[] letters = { 'H', 'e', 'l', 'l','o' };
            string greetings = new string(letters);
            Console.WriteLine("Greetings: {0}", greetings);

            //方法返回字符串
            string[] sarray = { "Hello", "From", "Tutorials", "Point" };
            string message = String.Join(" ", sarray);
            Console.WriteLine("Message: {0}", message);

            //用于转化值的格式化方法
            DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
            string chat = String.Format("Message sent at {0:t} on {0:D}",
            waiting);
            Console.WriteLine("Message: {0}", chat);
            Console.ReadKey() ;
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, October 10, 2012
```

String 类的属性

String 类有以下两个属性：

名称	描述
Chars	在当前 *String* 对象中获取 *Char* 对象的指定位置。
Length	在当前的 *String* 对象中获取字符数。

String 类的方法

String 类有许多方法用于 string 对象的操作。下面的表格提供了一些最常用的方法：

名称	描述
public static int Compare(string strA, string strB)	比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。该方法区分大小写。
public static int Compare(string strA, string strB, bool ignoreCase)	比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。但是，如果布尔参数为真时，该方法不区分大小写。
public static string Concat(string str0, string str1)	连接两个 string 对象。
public static string Concat(string str0, string str1, string str2)	连接三个 string 对象。
public static string Concat(string str0, string str1, string str2, string str3)	连接四个 string 对象。
public bool Contains(string value)	返回一个表示指定 string 对象是否出现在字符串中的值。
public static string Copy(string str)	创建一个与指定字符串具有相同值的新的 String 对象。
public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)	从 string 对象的指定位置开始复制指定数量的字符到 Unicode 字符数组中的指定位置。
public bool EndsWith(	

string value)	
public bool Equals(string value)	判断当前的 string 对象是否与指定的 string 对象具有相同的值。
public static bool Equals(string a, string b)	判断两个指定的 string 对象是否具有相同的值。
public static string Format(string format, Object arg0)	把指定字符串中一个或多个格式项替换为指定对象的字符串表示形式。
public int IndexOf(char value)	返回指定 Unicode 字符在当前字符串中第一次出现的索引，索引从 0 开始。
public int IndexOf(string value)	返回指定字符串在该实例中第一次出现的索引，索引从 0 开始。
public int IndexOf(char value, int startIndex)	返回指定 Unicode 字符从该字符串中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
public int IndexOf(string value, int startIndex)	返回指定字符串从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
public int IndexOfAny(char[] anyOf)	返回某一个指定的 Unicode 字符数组中任意字符在该实例中第一次出现的索引，索引从 0 开始。
public int IndexOfAny(char[] anyOf, int startIndex)	返回某一个指定的 Unicode 字符数组中任意字符从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
public string Insert(int startIndex, string value)	返回一个新的字符串，其中，指定的字符串被插入在当前 string 对象的指定索引位置。
public static bool IsNullOrEmpty(string value)	指示指定的字符串是否为 null 或者是否为一个空的字符串。
public static string Join(string separator, params string[] value)	连接一个字符串数组中的所有元素，使用指定的分隔符分隔每个元素。
public static string Join(string separator, string[] value, int startIndex, int count)	链接一个字符串数组中的指定元素，使用指定的分隔符分隔每个元素。
public int LastIndexOf(char value)	返回指定 Unicode 字符在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。
public int LastIndexOf(string value)	返回指定字符串在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。
public string Remove(int startIndex)	移除当前实例中的所有字符，从指定位置开始，一直到最后一个位置为止，并返回字符串。
public string Remove(int startIndex, int count)	从当前字符串的指定位置开始移除指定数量的字符，并返回字符串。
public string Replace(char	把当前 string 对象中，所有指定的 Unicode 字符替换为

public string Replace(char oldChar, char newChar)	把当前 string 对象中，所有指定的 Unicode 字符替换为另一个指定的 Unicode 字符，并返回新的字符串。
public string Replace(string oldValue, string newValue)	把当前 string 对象中，所有指定的字符串替换为另一个指定的字符串，并返回新的字符串。
public string[] Split(params char[] separator)	返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。
public string[] Split(char[] separator, int count)	返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。int 参数指定要返回的子字符串的最大数目。
public bool StartsWith(string value)	判断字符串实例的开头是否匹配指定的字符串。
public char[] ToCharArray()	返回一个带有当前 string 对象中所有字符的 Unicode 字符数组。
public char[] ToCharArray(int startIndex, int length)	返回一个带有当前 string 对象中所有字符的 Unicode 字符数组，从指定的索引开始，直到指定的长度为止。
public string ToLower()	把字符串转换为小写并返回。
public string ToUpper()	把字符串转换为大写并返回。
public string Trim()	移除当前 String 对象中的所有前导空白字符和后置空白字符。

上面的方法列表并不详尽，请访问 MSDN 库，查看完整的方法列表和 String 类构造函数。

实例

下面的实例演示了上面提到的一些方法：

比较字符串

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str1 = "This is test";
         string str2 = "This is text";

         if (String.Compare(str1, str2) == 0)
         {
            Console.WriteLine(str1 + " and " + str2 +  " are equal.");
         }
         else
         {
            Console.WriteLine(str1 + " and " + str2 + " are not equal.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is test and This is text are not equal.
```

字符串包含字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str = "This is test";
         if (str.Contains("test"))
         {
            Console.WriteLine("The sequence 'test' was found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The sequence 'test' was found.
```

获取子字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str = "Last night I dreamt of San Pedro";
         Console.WriteLine(str);
         string substr = str.Substring(23);
         Console.WriteLine(substr);
      }
      Console.ReadKey() ;
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
San Pedro
```

连接字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string[] starray = new string[]{"Down the way nights are dark",
         "And the sun shines daily on the mountain top",
         "I took a trip on a sailing ship",
         "And when I reached Jamaica",
         "I made a stop"};

         string str = String.Join("\n", starray);
         Console.WriteLine(str);
      }
      Console.ReadKey() ;
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

C# 结构（Struct）

在 C# 中，结构是值类型数据结构。它使得一个单一变量可以存储各种数据类型的相关数据。**struct** 关键字用于创建结构。

结构是用来代表一个记录。假设您想跟踪图书馆中书的动态。您可能想跟踪每本书的以下属性：

- Title
- Author
- Subject
- Book ID

定义结构

为了定义一个结构，您必须使用 struct 语句。struct 语句为程序定义了一个带有多个成员的新的数据类型。

例如，您可以按照如下的方式声明 Book 结构：

```
struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

下面的程序演示了结构的用法：

```
using System;

struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};

public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1;        /* 声明 Book1，类型为 Book */
      Books Book2;        /* 声明 Book2，类型为 Book */

      /* book 1 详述 */
      Book1.title = "C Programming";
      Book1.author = "Nuha Ali";
      Book1.subject = "C Programming Tutorial";
      Book1.book_id = 6495407;

      /* book 2 详述 */
      Book2.title = "Telecom Billing";
      Book2.author = "Zara Ali";
      Book2.subject =  "Telecom Billing Tutorial";
      Book2.book_id = 6495700;

      /* 打印 Book1 信息 */
      Console.WriteLine( "Book 1 title : {0}", Book1.title);
      Console.WriteLine("Book 1 author : {0}", Book1.author);
      Console.WriteLine("Book 1 subject : {0}", Book1.subject);
      Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

      /* 打印 Book2 信息 */
      Console.WriteLine("Book 2 title : {0}", Book2.title);
      Console.WriteLine("Book 2 author : {0}", Book2.author);
      Console.WriteLine("Book 2 subject : {0}", Book2.subject);
      Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

      Console.ReadKey();

   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

C# 结构的特点

您已经用了一个简单的名为 Books 的结构。在 C# 中的结构与传统的 C 或 C++ 中的结构不同。C# 中的结构有一下特点：

- 结构可带有方法、字段、索引、属性、运算符方法和事件。
- 结构可定义构造函数，但不能定义析构函数。但是，您不能为结构定义默认的构造函数。默认的构造函数是自动定义的，且不能被改变。
- 与类不同，结构不能继承其他的结构或类。
- 结构不能作为其他结构或类的基础结构。
- 结构可实现一个或多个接口。
- 结构成员不能指定为 abstract、virtual 或 protected。
- 当您使用 **New** 操作符创建一个结构对象时，会调用适当的构造函数来创建结构。与类不同，结构可以不使用 New 操作符即可被实例化。
- 如果不使用 New 操作符，只有在所有的字段都被初始化之后，字段才被赋值，对象才被使用。

类 vs 结构

类和结构有以下几个基本的不同点：

- 类是引用类型，结构是值类型。
- 结构不支持继承。
- 结构不能声明默认的构造函数。

针对上述讨论，让我们重写前面的实例：

```
using System;

struct Books
{
   private string title;
   private string author;
   private string subject;
   private int book_id;
   public void getValues(string t, string a, string s, int id)
   {
      title = t;
      author = a;
      subject = s;
      book_id = id;
   }
   public void display()
   {
      Console.WriteLine("Title : {0}", title);
      Console.WriteLine("Author : {0}", author);
      Console.WriteLine("Subject : {0}", subject);
      Console.WriteLine("Book_id :{0}", book_id);
   }

};

public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1 = new Books(); /* 声明 Book1，类型为 Book */
      Books Book2 = new Books(); /* 声明 Book2，类型为 Book */

      /* book 1 详述 */
      Book1.getValues("C Programming",
      "Nuha Ali", "C Programming Tutorial",6495407);

      /* book 2 详述 */
      Book2.getValues("Telecom Billing",
      "Zara Ali", "Telecom Billing Tutorial", 6495700);

      /* 打印 Book1 信息 */
      Book1.display();

      /* 打印 Book2 信息 */
      Book2.display();

      Console.ReadKey();

   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

C# 枚举（Enum）

枚举是一组命名整型常量。枚举类型是使用 **enum** 关键字声明的。

C# 枚举是值数据类型。换句话说，枚举包含自己的值，且不能继承或传递继承。

声明 *enum* 变量

声明枚举的一般语法：

```
enum <enum_name>
{
    enumeration list
};
```

其中，

- *enum_name* 指定枚举的类型名称。
- *enumeration list* 是一个用逗号分隔的标识符列表。

枚举列表中的每个符号代表一个整数值，一个比它前面的符号大的整数值。默认情况下，第一个枚举符号的值是 0.例如：

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

实例

下面的实例演示了枚举变量的用法：

```
using System;
namespace EnumApplication
{
   class EnumProgram
   {
      enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

      static void Main(string[] args)
      {
         int WeekdayStart = (int)Days.Mon;
         int WeekdayEnd = (int)Days.Fri;
         Console.WriteLine("Monday: {0}", WeekdayStart);
         Console.WriteLine("Friday: {0}", WeekdayEnd);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Monday: 1
Friday: 5
```

C# 类（Class）

当您定义一个类时，您定义了一个数据类型的蓝图。这实际上并没有定义任何的数据，但它定义了类的名称意味着什么，也就是说，类的对象由什么组成及在这个对象上可执行什么操作。对象是类的实例。构成类的方法和变量成为类的成员。

类的定义

类的定义是以关键字 **class** 开始，后跟类的名称。类的主体，包含在一对花括号内。下面是类定义的一般形式：

```
<access specifier> class  class_name
{
    // member variables
    <access specifier> <data type> variable1;
    <access specifier> <data type> variable2;
    ...
    <access specifier> <data type> variableN;
    // member methods
    <access specifier> <return type> method1(parameter_list)
    {
        // method body
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}
```

请注意：

- 访问标识符 <access specifier> 指定了对类及其成员的访问规则。如果没有指定，则使用默认的访问标识符。类的默认访问标识符是 **internal**，成员的默认访问标识符是 **private**。
- 数据类型 <data type> 指定了变量的类型，返回类型 <return type> 指定了返回的方法返回的数据类型。
- 如果要访问类的成员，您要使用点（.）运算符。
- 点运算符链接了对象的名称和成员的名称。

下面的实例说明了目前为止所讨论的概念：

```
using System;
namespace BoxApplication
{
    class Box
    {
       public double length;   // 长度
       public double breadth;  // 宽度
       public double height;   // 高度
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // 声明 Box1，类型为 Box
            Box Box2 = new Box();        // 声明 Box2，类型为 Box
            double volume = 0.0;         // 体积

            // Box1 详述
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // Box2 详述
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // Box1 的体积
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Box1 的体积： {0}",  volume);

            // Box2 的体积
            volume = Box2.height * Box2.length * Box2.breadth;
            Console.WriteLine("Box2 的体积： {0}", volume);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
```

成员函数和封装

类的成员函数是一个在类定义中有它的定义或原型的函数，就像其他变量一样。作为类的一个成为，它能在类的任何对象上操作，且能访问该对象的类的所有成员。

成员变量是对象的属性（从设计角度），且它们保持私有来实现封装。这些变量只能使用公共成员函数来访问。

让我们使用上面的概念来设置和获取一个类中不同的类成员的值：

```
using System;
namespace BoxApplication
{
    class Box
    {
       private double length;   // 长度
       private double breadth;  // 宽度
       private double height;   // 高度
       public void setLength( double len )
       {
            length = len;
       }

       public void setBreadth( double bre )
       {
            breadth = bre;
       }

       public void setHeight( double hei )
       {
            height = hei;
       }
       public double getVolume()
       {
           return length * breadth * height;
       }
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // 声明 Box1，类型为 Box
            Box Box2 = new Box();         // 声明 Box2，类型为 Box
            double volume;                 // 体积

            // Box1 详述
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // Box2 详述
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}" ,volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
```

C# 中的构造函数

类的 构造函数 是类的一个特殊的成员函数，当创建类的新对象时执行。

构造函数的名称与类的名称完全相同，它没有任何返回类型。

下面的实例说明了构造函数的概念：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line()
      {
         Console.WriteLine("对象已创建");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建
线条的长度： 6
```

默认的构造函数没有任何参数。但是如果您需要一个带有参数的构造函数可以有参数，这种构造函数叫做参数化构造函数。这种技术可以帮助您在创建对象的同时给对象赋初始值，具体请看下面实例：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line(double len)  // 参数化构造函数
      {
         Console.WriteLine("对象已创建，length = {0}", len);
         length = len;
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line(10.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建，length = 10
线条的长度： 10
线条的长度： 6
```

C# 中的析构函数

类的 析构函数 是类的一个特殊的成员函数，当类的对象超出范围时执行。

析构函数的名称是在类的名称前加上一个波浪形（~）作为前缀，它不返回值，也不带任何参数。

析构函数用于在结束程序（比如关闭文件、释放内存等）之前释放资源。析构函数不能继承或重载。

下面的实例说明了析构函数的概念：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line()  // 构造函数
      {
         Console.WriteLine("对象已创建");
      }
      ~Line() //析构函数
      {
         Console.WriteLine("对象已删除");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建
线条的长度： 6
对象已删除
```

C# 类的静态成员

我们可以使用 **static** 关键字把类成员定义为静态的。当我们声明一个类成员为静态时，意味着无论有多少个类的对象被创建，只会有一个该静态成员的副本。

关键字 **static** 意味着类中只有一个该成员的实例。静态变量用于定义常量，因为它们的值可以通过直接调用类而不需要创建类的实例来获取。静态变量可在成员函数或类的定义外部进行初始化。您也可以在类的定义内部初始化静态变量。

下面的实例演示了静态变量的用法：

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
       public static int num;
        public void count()
        {
            num++;
        }
        public int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s1 = new StaticVar();
            StaticVar s2 = new StaticVar();
            s1.count();
            s1.count();
            s1.count();
            s2.count();
            s2.count();
            s2.count();
            Console.WriteLine("s1 的变量 num： {0}", s1.getNum());
            Console.WriteLine("s2 的变量 num： {0}", s2.getNum());
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
s1 的变量 num： 6
s2 的变量 num： 6
```

您也可以把一个成员函数声明为 **static**。这样的函数只能访问静态变量。静态函数在对象被创建之前就已经存在。下面的实例演示了静态函数的用法：

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
       public static int num;
        public void count()
        {
            num++;
        }
        public static int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s = new StaticVar();
            s.count();
            s.count();
            s.count();
            Console.WriteLine("变量 num： {0}", StaticVar.getNum());
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
变量 num： 3
```

C# 继承

继承是面向对象程序设计中最重要的概念之一。继承允许我们根据一个类来定义另一个类来定义一个类，这使得创建和维护应用程序变得更容易。同时也有利于重用代码和节省开发时间。

当创建一个类时，程序员不需要完全重新编写新的数据成员和成员函数，只需要设计一个新的类，继承了已有的类的成员即可。这个已有的类被称为的基类，这个新的类被称为派生类。

继承的思想实现了 属于（**IS-A**） 关系。例如，哺乳动物 属于（**IS-A**） 动物，狗 属于（**IS-A**） 哺乳动物，因此狗 属于（**IS-A**） 动物。

基类和派生类

一个类可以派生自多个类或接口，这意味着它可以从多个基类或接口继承数据和函数。

C# 中创建派生类的语法如下：

```
<acess-specifier> class <base_class>
{
 ...
}
class <derived_class> : <base_class>
{
 ...
}
```

假设，有一个基类 Shape，它的派生类是 Rectangle：

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // 派生类
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }

   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         // 打印对象的面积
         Console.WriteLine("总面积： {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
总面积： 35
```

基类的初始化

派生类继承了基类的成员变量和成员方法。因此父类对象应在子类对象创建之前被创建。您可以在成员初始化列表中进行父类的初始化。

下面的程序演示了这点：

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      // 成员变量
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }
      public double GetArea()
      {
         return length * width;
      }
      public void Display()
      {
         Console.WriteLine("长度： {0}", length);
         Console.WriteLine("宽度： {0}", width);
         Console.WriteLine("面积： {0}", GetArea());
      }
   }//end class Rectangle
   class Tabletop : Rectangle
   {
      private double cost;
      public Tabletop(double l, double w) : base(l, w)
      { }
      public double GetCost()
      {
         double cost;
         cost = GetArea() * 70;
         return cost;
      }
      public void Display()
      {
         base.Display();
         Console.WriteLine("成本： {0}", GetCost());
      }
   }
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
长度： 4.5
宽度： 7.5
面积： 33.75
成本： 2362.5
```

C# 多重继承

C# 不支持多重继承。但是，您可以使用接口来实现多重继承。下面的程序演示了这点：

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // 基类 PaintCost
   public interface PaintCost
   {
      int getCost(int area);

   }
   // 派生类
   class Rectangle : Shape, PaintCost
   {
      public int getArea()
      {
         return (width * height);
      }
      public int getCost(int area)
      {
         return area * 70;
      }
   }
   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();
         int area;
         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();
         // 打印对象的面积
         Console.WriteLine("总面积： {0}",  Rect.getArea());
         Console.WriteLine("油漆总成本： ${0}" , Rect.getCost(area));
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
总面积： 35
油漆总成本： $2450
```

C# 多态性

多态性意味着有多重形式。在面向对象编程范式中，多态性往往表现为"一个接口，多个功能"。

多态性可以是静态的或动态的。在静态多态性中，函数的响应是在编译时发生的。在动态多态性中，函数的响应是在运行时发生的。

静态多态性

在编译时，函数和对象的连接机制被称为早期绑定，也被称为静态绑定。C# 提供了两种技术来实现静态多态性。分别为：

- 函数重载
- 运算符重载

运算符重载将在下一章节讨论，接下来我们将讨论函数重载。

函数重载

您可以在同一个范围内对相同的函数名有多个定义。函数的定义必须彼此不同，可以是参数列表中的参数类型不同，也可以是参数个数不同。不能重载只有返回类型不同的函数声明。

下面的实例演示了几个相同的函数 **print()**，用于打印不同的数据类型：

```
using System;
namespace PolymorphismApplication
{
   class Printdata
   {
      void print(int i)
      {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f)
      {
         Console.WriteLine("Printing float: {0}" , f);
      }

      void print(string s)
      {
         Console.WriteLine("Printing string: {0}", s);
      }
      static void Main(string[] args)
      {
         Printdata p = new Printdata();
         // 调用 print 来打印整数
         p.print(5);
         // 调用 print 来打印浮点数
         p.print(500.263);
         // 调用 print 来打印字符串
         p.print("Hello C++");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

动态多态性

C# 允许您使用关键字 **abstract** 创建抽象类，用于提供接口的部分类的实现。当一个派生类继承自该抽象类时，实现即完成。抽象类包含抽象方法，抽象方法可被派生类实现。派生类具有更专业的功能。

请注意，下面是有关抽象类的一些规则：

- 您不能创建一个抽象类的实例。
- 您不能在一个抽象类外部声明一个抽象方法。
- 通过在类定义前面放置关键字 **sealed**，可以将类声明为密封类。当一个类被声明为 **sealed** 时，它不能被继承。抽象类不能被声明为 sealed。

下面的程序演示了一个抽象类：

```
using System;
namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length = a;
         width = b;
      }
      public override int area ()
      {
         Console.WriteLine("Rectangle 类的面积：");
         return (width * length);
      }
   }

   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle r = new Rectangle(10, 7);
         double a = r.area();
         Console.WriteLine("面积： {0}",a);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Rectangle 类的面积：
面积： 70
```

当有一个定义在类中的函数需要在继承类中实现时，可以使用虚方法。虚方法是使用关键字 **virtual** 声明的。虚方法可以在不同的继承类中有不同的实现。对虚方法的调用是在运行时发生的。

动态多态性是通过 抽象类 和 虚方法 实现的。

下面的程序演示了这点：

```
using System;
namespace PolymorphismApplication
{
   class Shape
   {
      protected int width, height;
      public Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      public virtual int area()
      {
         Console.WriteLine("父类的面积：");
         return 0;
      }
   }
   class Rectangle: Shape
   {
      public Rectangle( int a=0, int b=0): base(a, b)
      {

      }
      public override int area ()
      {
         Console.WriteLine("Rectangle 类的面积：");
         return (width * height);
      }
   }
   class Triangle: Shape
   {
      public Triangle(int a = 0, int b = 0): base(a, b)
      {

      }
      public override int area()
      {
         Console.WriteLine("Triangle 类的面积：");
         return (width * height / 2);
      }
   }
   class Caller
   {
      public void CallArea(Shape sh)
      {
         int a;
         a = sh.area();
         Console.WriteLine("面积： {0}", a);
      }
   }
   class Tester
   {

      static void Main(string[] args)
      {
         Caller c = new Caller();
         Rectangle r = new Rectangle(10, 7);
         Triangle t = new Triangle(10, 5);
         c.CallArea(r);
         c.CallArea(t);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Rectangle 类的面积：
面积：70
Triangle 类的面积：
面积：25
```

C# 运算符重载

您可以重定义或重载 C# 中内置的运算符。因此，程序员也可以使用用户自定义类型的运算符。重载运算符是具有特殊名称的函数，是通过关键字 **operator** 后跟运算符的符号来定义的。与其他函数一样，重载运算符有返回类型和参数列表。

例如，请看下面的函数：

```
public static Box operator+ (Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

上面的函数为用户自定义的类 Box 实现了加法运算符（+）。它把两个 Box 对象的属性相加，并返回相加后的 Box 对象。

运算符重载的实现

下面的程序演示了完整的实现：

```
using System;

namespace OperatorOvlApplication
{
   class Box
   {
      private double length;      // 长度
      private double breadth;     // 宽度
      private double height;      // 高度

      public double getVolume()
      {
         return length * breadth * height;
      }
      public void setLength( double len )
      {
         length = len;
      }

      public void setBreadth( double bre )
      {
         breadth = bre;
      }

      public void setHeight( double hei )
      {
         height = hei;
      }
      // 重载 + 运算符来把两个 Box 对象相加
      public static Box operator+ (Box b, Box c)
      {
         Box box = new Box();
```

```
            box.length = b.length + c.length;
            box.breadth = b.breadth + c.breadth;
            box.height = b.height + c.height;
            return box;
        }

    }

    class Tester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();         // 声明 Box1，类型为 Box
            Box Box2 = new Box();         // 声明 Box2，类型为 Box
            Box Box3 = new Box();         // 声明 Box3，类型为 Box
            double volume = 0.0;          // 体积

            // Box1 详述
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // Box2 详述
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}", volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            // 把两个对象相加
            Box3 = Box1 + Box2;

            // Box3 的体积
            volume = Box3.getVolume();
            Console.WriteLine("Box3 的体积： {0}", volume);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
Box3 的体积： 5400
```

可重载和不可重载运算符

下表描述了 C# 中运算符重载的能力：

运算符	描述
+, -, !, ~, ++, --	这些一元运算符只有一个操作数，且可以被重载。
+, -, *, /, %	这些二元运算符带有两个操作数，且可以被重载。
==, !=, <, >, <=, >=	这些比较运算符可以被重载。
&&, ||	这些条件逻辑运算符不能被直接重载。
+=, -=, *=, /=, %=	这些赋值运算符不能被重载。
=, ., ?:, ->, new, is, sizeof, typeof	这些运算符不能被重载。

实例

针对上述讨论，让我们扩展上面的实例，重载更多的运算符：

```
using System;

namespace OperatorOvlApplication
{
    class Box
    {
       private double length;      // 长度
       private double breadth;     // 宽度
       private double height;      // 高度

       public double getVolume()
       {
         return length * breadth * height;
       }
      public void setLength( double len )
      {
          length = len;
      }

      public void setBreadth( double bre )
      {
          breadth = bre;
      }

      public void setHeight( double hei )
      {
          height = hei;
      }
      // 重载 + 运算符来把两个 Box 对象相加
      public static Box operator+ (Box b, Box c)
      {
          Box box = new Box();
          box.length = b.length + c.length;
          box.breadth = b.breadth + c.breadth;
          box.height = b.height + c.height;
          return box;
      }

      public static bool operator == (Box lhs, Box rhs)
      {
          bool status = false;
          if (lhs.length == rhs.length && lhs.height == rhs.height
             && lhs.breadth == rhs.breadth)
          {
              status = true;
          }
```

```
        return status;
    }
    public static bool operator !=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length != rhs.length || lhs.height != rhs.height
            || lhs.breadth != rhs.breadth)
        {
            status = true;
        }
        return status;
    }
    public static bool operator <(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length < rhs.length && lhs.height
            < rhs.height && lhs.breadth < rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator >(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length > rhs.length && lhs.height
            > rhs.height && lhs.breadth > rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator <=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length <= rhs.length && lhs.height
            <= rhs.height && lhs.breadth <= rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator >=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length >= rhs.length && lhs.height
           >= rhs.height && lhs.breadth >= rhs.breadth)
        {
            status = true;
        }
        return status;
    }
    public override string ToString()
    {
        return String.Format("({0}, {1}, {2})", length, breadth, height);
    }

 }

 class Tester
 {
    static void Main(string[] args)
    {
      Box Box1 = new Box();          // 声明 Box1，类型为 Box
      Box Box2 = new Box();          // 声明 Box2，类型为 Box
      Box Box3 = new Box();          // 声明 Box3，类型为 Box
      Box Box4 = new Box();
      double volume = 0.0;   // 体积
```

```
            // Box1 详述
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // Box2 详述
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

           // 使用重载的 ToString() 显示两个盒子
            Console.WriteLine("Box1： {0}", Box1.ToString());
            Console.WriteLine("Box2： {0}", Box2.ToString());

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}", volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            // 把两个对象相加
            Box3 = Box1 + Box2;
            Console.WriteLine("Box3： {0}", Box3.ToString());
            // Box3 的体积
            volume = Box3.getVolume();
            Console.WriteLine("Box3 的体积： {0}", volume);

            //comparing the boxes
            if (Box1 > Box2)
              Console.WriteLine("Box1 大于 Box2");
            else
              Console.WriteLine("Box1 不大于 Box2");
            if (Box1 < Box2)
              Console.WriteLine("Box1 小于 Box2");
            else
              Console.WriteLine("Box1 不小于 Box2");
            if (Box1 >= Box2)
              Console.WriteLine("Box1 大于等于 Box2");
            else
              Console.WriteLine("Box1 不大于等于 Box2");
            if (Box1 <= Box2)
              Console.WriteLine("Box1 小于等于 Box2");
            else
              Console.WriteLine("Box1 不小于等于 Box2");
            if (Box1 != Box2)
              Console.WriteLine("Box1 不等于 Box2");
            else
              Console.WriteLine("Box1 等于 Box2");
            Box4 = Box3;
            if (Box3 == Box4)
              Console.WriteLine("Box3 等于 Box4");
            else
              Console.WriteLine("Box3 不等于 Box4");

            Console.ReadKey();
          }
      }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1： (6, 7, 5)
Box2： (12, 13, 10)
Box1 的体积： 210
Box2 的体积： 1560
Box3： (18, 20, 15)
Box3 的体积： 5400
Box1 不大于 Box2
Box1 小于 Box2
Box1 不大于等于 Box2
Box1 小于等于 Box2
Box1 不等于 Box2
Box3 等于 Box4
```

C# 接口（Interface）

接口定义了所有类继承接口时应遵循的语法合同。接口定义了语法合同 **"是什么"** 部分，派生类定义了语法合同 **"怎么做"** 部分。

接口定义了属性、方法和事件，这些都是接口的成员。接口只包含了成员的声明。成员的定义是派生类的责任。接口提供了派生类应遵循的标准结构。

抽象类在某种程度上与接口类似，但是，它们大多只是用在当只有少数方法由基类声明由派生类实现时。

声明接口

接口使用 **interface** 关键字声明，它与类的声明类似。接口声明默认是 public 的。下面是一个接口声明的实例：

```
public interface ITransactions
{
   // 接口成员
   void showTransaction();
   double getAmount();
}
```

实例

下面的实例演示了上面接口的实现：

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceApplication
{

   public interface ITransactions
   {
      // 接口成员
      void showTransaction();
      double getAmount();
   }
   public class Transaction : ITransactions
   {
      private string tCode;
      private string date;
      private double amount;
      public Transaction()
      {
         tCode = " ";
         date = " ";
         amount = 0.0;
      }
      public Transaction(string c, string d, double a)
      {
         tCode = c;
         date = d;
         amount = a;
      }
      public double getAmount()
      {
         return amount;
      }
      public void showTransaction()
      {
         Console.WriteLine("Transaction: {0}", tCode);
         Console.WriteLine("Date: {0}", date);
         Console.WriteLine("Amount: {0}", getAmount());

      }

   }
   class Tester
   {
      static void Main(string[] args)
      {
         Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
         Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
         t1.showTransaction();
         t2.showTransaction();
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

C# 命名空间（Namespace）

命名空间的设计目的是为了提供一种让一组名称与其他名称分隔开的方式。在一个命名空间中声明的类的名称与另一个命名空间中声明的相同的类的名称不冲突。

定义命名空间

命名空间的定义是以关键字 **namespace** 开始，后跟命名空间的名称，如下所示：

```
namespace namespace_name
{
   // 代码声明
}
```

为了调用支持命名空间版本的函数或变量，会把命名空间的名称置于前面，如下所示：

```
namespace_name.item_name;
```

下面的程序演示了命名空间的用法：

```
using System;
namespace first_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
class TestClass
{
   static void Main(string[] args)
   {
      first_space.namespace_cl fc = new first_space.namespace_cl();
      second_space.namespace_cl sc = new second_space.namespace_cl();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

using 关键字

using 关键字表明程序使用的是给定命名空间中的名称。例如，我们在程序中使用 **System** 命名空间，其中定义了类 Console。我们可以只写：

```
Console.WriteLine ("Hello there");
```

我们可以写完全限定名称，如下：

```
System.Console.WriteLine("Hello there");
```

您也可以使用 **using** 命名空间指令，这样在使用的时候就不用在前面加上命名空间名称。该指令告诉编译器随后的代码使用了指定命名空间中的名称。下面的代码延时了命名空间的应用。

让我们使用 using 指定重写上面的实例：

```
using System;
using first_space;
using second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class efg
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
class TestClass
{
   static void Main(string[] args)
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

嵌套命名空间

命名空间可以被嵌套，即您可以在一个命名空间内定义另一个命名空间，如下所示：

```
namespace namespace_name1
{
   // 代码声明
   namespace namespace_name2
   {
     // 代码声明
   }
}
```

您可以使用点（.）运算符访问嵌套的命名空间的成员，如下所示：

```
using System;
using first_space;
using first_space.second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
   namespace second_space
   {
      class efg
      {
         public void func()
         {
            Console.WriteLine("Inside second_space");
         }
      }
   }
}

class TestClass
{
   static void Main(string[] args)
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

C# 预处理器指令

预处理器指令指导编译器在实际编译开始之前对信息进行预处理。

所有的预处理器指令都是以 # 开始。且在一行上，只有空白字符可以出现在预处理器指令之前。预处理器指令不是语句，所以它们不以分号（;）结束。

C# 编译器没有一个单独的预处理器，但是，指令被处理时就像是有一个单独的预处理器一样。在 C# 中，预处理器指令用于在条件编译中起作用。与 C 和 C++ 不同指令不用，它们不是用来创建宏。一个预处理器指令必须是该行上的唯一指令。

C# 预处理器指令列表

下表列出了 C# 中可用的预处理器指令：

预处理器指令	描述
#define	它用于定义一系列成为符号的字符。
#undef	它用于取消定义符号。
#if	它用于测试符号是否为真。
#else	它用于创建复合条件指令，与 #if 一起使用。
#elif	它用于创建复合条件指令。
#endif	指定一个条件指令的结束。
#line	它可以让您修改编译器的行数以及（可选地）输出错误和警告的文件名。
#error	它允许从代码的指定位置生成一个错误。
#warning	它允许从代码的指定位置生成一级警告。
#region	它可以让您在使用 Visual Studio Code Editor 的大纲特性时，指定一个可展开或折叠的代码块。
#endregion	它标识着 #region 块的结束。

#define 预处理器

define 预处理器指令创建符号常量。

define 允许您定义一个符号，这样，通过使用符号作为传递给 #if 指令的表达式，表达式将返回 true。它的语法如下：

```
#define symbol
```

下面的程序说明了这点：

```
#define PI
using System;
namespace PreprocessorDAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         #if (PI)
            Console.WriteLine("PI is defined");
         #else
            Console.WriteLine("PI is not defined");
         #endif
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
PI is defined
```

条件指令

您可以使用 #if 指令来创建一个条件指令。条件指令用于测试符号是否为真。如果为真，编译器会执行 #if 和下一个指令之间的代码。

条件指令的语法：

```
#if symbol [operator symbol]...
```

其中，*symbol* 是要测试的符号名称。您也可以使用 true 和 false，或在符号前放置否定运算符。

运算符符号是用于评价符号的运算符。可以运算符可以是下列运算符之一：

- == (equality)
- != (inequality)
- && (and)

- || (or)

您也可以用括号把符号和运算符进行分组。条件指令用于在调试版本或编译指定配置时编译代码。一个以 **#if** 指令开始的条件指令，必须显示地以一个 **#endif** 指令终止。

下面的程序演示了条件指令的用法：

```
#define DEBUG
#define VC_V10
using System;
public class TestClass
{
   public static void Main()
   {

      #if (DEBUG && !VC_V10)
         Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && VC_V10)
         Console.WriteLine("VC_V10 is defined");
      #elif (DEBUG && VC_V10)
         Console.WriteLine("DEBUG and VC_V10 are defined");
      #else
         Console.WriteLine("DEBUG and VC_V10 are not defined");
      #endif
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
DEBUG and VC_V10 are defined
```

C# 正则表达式

正则表达式 是一种匹配输入文本的模式。.Net 框架提供了允许这种匹配的正则表达式引擎。模式由一个或多个字符、运算符和结构组成。

定义正则表达式

下面列出了用于定义正则表达式的各种类别的字符、运算符和结构。

- 字符转义
- 字符类
- 定位点
- 分组构造
- 限定符
- 反向引用构造
- 备用构造
- 替换
- 杂项构造

字符转义

正则表达式中的反斜杠字符（\）指示其后跟的字符是特殊字符，或应按原义解释该字符。

下表列出了转义字符：

转义字符	描述	模式	匹配
\a	与报警 (bell) 符 \u0007 匹配。	\a	"Warning!" + '\u0007' 中的 "\u0007"
\b	在字符类中，与退格键 \u0008 匹配。	[\b]{3,}	"\b\b\b\b" 中的 "\b\b\b\b"
\t	与制表符 \u0009 匹配。	(\w+)\t	"Name\tAddr\t" 中的 "Name\t" 和 "Addr\t"
\r	与回车符 \u000D 匹配。（\r 与换行符 \n 不是等效的。）	\r\n(\w+)	"\r\Hello\nWorld." 中的 "\r\nHello"
\v	与垂直制表符 \u000B 匹配。	[\v]{2,}	"\v\v\v" 中的 "\v\v\v"
\f	与换页符 \u000C 匹配。	[\f]{2,}	"\f\f\f" 中的 "\f\f\f"
\n	与换行符 \u000A 匹配。	\r\n(\w+)	"\r\Hello\nWorld." 中的 "\r\nHello"
\e	与转义符 \u001B 匹配。	\e	"\x001B" 中的 "\x001B"
\ nnn	使用八进制表示形式指定一个字符（nnn 由二到三位数字组成）。	\w\040\w	"a bc d" 中的 "a b" 和 "c d"
\x nn	使用十六进制表示形式指定字符（nn 恰好由两位数字组成）。	\w\x20\w	"a bc d" 中的 "a b" 和 "c d"
\c X **\c x**	匹配 X 或 x 指定的 ASCII 控件字符，其中 X 或 x 是控件字符的字母。	\cC	"\x0003" 中的 "\x0003" (Ctrl-C)
\u nnnn	使用十六进制表示形式匹配一个 Unicode 字符（由 nnnn 表示的四位数）。	\w\u0020\w	"a bc d" 中的 "a b" 和 "c d"
****	在后面带有不识别的转义字符时，与该字符匹配。	\d+[+-x*]\d+\d+[+-x*\d+	"(2+2) *39" 中的 "2+2" 和 "3*9"

字符类

字符类与一组字符中的任何一个字符匹配。 下表列出了字符类：

字符类	描述	模式	匹配
[character_group]	匹配 character_group 中的任何单个字符。 默认情况下，匹配区分大小写。	[mn]	"mat" 中的 "m"， "moon" 中的 "m" 和 "n"
[^character_group]	非：与不在 character_group 中的任何单个字符匹配。 默认情况下，character_group 中的字符区分大小写。	[^aei]	"avail" 中的 "v" 和 "l"
[first - last]	字符范围：与从 first 到 last 的范围中的任何单个字符匹配。	(\w+)\t	"Name\tAddr\t" 中的 "Name\t" 和 "Addr\t"
.	通配符：与除 \n 之外的任何单个字符匹配。 若要匹配原意句点字符（. 或 \u002E），您必须在该字符前面加上转义符 (.)。	a.e	"have" 中的 "ave"， "mate" 中的 "ate"
\p{ name }	与 *name* 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。	\p{Lu}	"City Lights" 中的 "C" 和 "L"
\P{ name }	与不在 *name* 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。	\P{Lu}	"City" 中的 "i"、 "t" 和 "y"
\w	与任何单词字符匹配。	\w	"Room#1" 中的 "R"、 "o"、 "m" 和 "1"
\W	与任何非单词字符匹配。	\W	"Room#1" 中的 "#"
\s	与任何空白字符匹配。	\w\s	"ID A1.3" 中的 "D "
\S	与任何非空白字符匹配。	\s\S	"int _ctr" 中的 " "
\d	与任何十进制数字匹配。	\d	"4 = IV" 中的 "4"
\D	匹配不是十进制数的任意字符。	\D	"4 = IV" 中的 " "、 "="、 " "、 "I" 和 "V"

定位点

定位点或原子零宽度断言会使匹配成功或失败，具体取决于字符串中的当前位置，但它们不会使引擎在字符串中前进或使用字符。 下表列出了定位点：

断言	描述	模式	匹配
^	匹配必须从字符串或一行的开头开始。	^\d{3}	"567-777-" 中的 "567"
$	匹配必须出现在字符串的末尾或出现在行或字符串末尾的 **\n** 之前。	-\d{4}$	"8-12-2012" 中的 "-2012"
\A	匹配必须出现在字符串的开头。	\A\w{3}	"Code-007-" 中的 "Code"
\Z	匹配必须出现在字符串的末尾或出现在字符串末尾的 **\n** 之前。	-\d{3}\Z	"Bond-901-007" 中的 "-007"
\z	匹配必须出现在字符串的末尾。	-\d{3}\z	"-901-333" 中的 "-333"
\G	匹配必须出现在上一个匹配结束的地方。	\G(\d)	"(1)(3)(5)7" 中的 "(1)"、 "(3)" 和 "(5)"
\b	匹配必须出现在 **\w**（字母数字）和 **\W**（非字母数字）字符之间的边界上。	\w	"Room#1" 中的 "R"、 "o"、 "m" 和 "1"
\B	匹配不得出现在 **\b** 边界上。	\Bend\w*\b	"end sends endure lender" 中的 "ends" 和 "ender"

分组构造

分组构造描述了正则表达式的子表达式，通常用于捕获输入字符串的子字符串。 下表列出了分组构造：

分组构造	描述	模式	匹配
(subexpression)	捕获匹配的子表达式并将其分配到一个从零开始的序号中。	(\w)\1	"deep" 中的 "ee"
(?< name >subexpression)	将匹配的子表达式捕获到一个命名组中。	(?< double>\w)\k< double>	"deep" 中的 "ee"
(?< name1 -name2 >subexpression)	定义平衡组定义。	(((?'Open'()\(\))+((?'Close-Open'))\(\))+)*(?(Open)(?!))$	"3+2^((1-3)(3-1))" 中的 "((1-3)(3-1))"
(?: subexpression)	定义非捕获组。	Write(?:Line)?	"Console.WriteLine()" 中的 "WriteLine"
(?imnsx-imnsx:subexpression)	应用或禁用 *subexpression* 中指定的选项。	A\d{2}(?i:\w+)\b	"A12xl A12XL a12xl" 中的 "A12xl" 和 "A12XL"
(?= subexpression)	零宽度正预测先行断言。	\w+(?=.)	"He is. The dog ran. The sun is out." 中的 "is"、 "ran" 和 "out"
(?! subexpression)	零宽度负预测先行断言。	\b(?!un)\w+\b	"unsure sure unity used" 中的 "sure" 和 "used"
(?< =subexpression)	零宽度正回顾后发断言。	(?< =19)\d{2}\b	"1851 1999 1950 1905 2003" 中的 "51" 和 "03"
(?< ! subexpression)	零宽度负回顾后发断言。	(?< !19)\d{2}\b	"end sends endure lender" 中的 "ends" 和 "ender"
(?> subexpression)	非回溯（也称为"贪婪"）子表达式。	[13579](?>A+B+)	"1ABB 3ABBC 5AB 5AC" 中的 "1ABB"、 "3ABB" 和 "5AB"

限定符

限定符指定在输入字符串中必须存在上一个元素（可以是字符、组或字符类）的多少个实例才能出现匹配项。 限定符包括下表中列出的语言元素。 下表列出了限定符：

限定符	描述	模式	匹配
*	匹配上一个元素零次或多次。	\d*.\d	".0"、 "19.9"、 "219.9"
+	匹配上一个元素一次或多次。	"be+"	"been" 中的 "bee"， "bent" 中的 "be"
?	匹配上一个元素零次或一次。	"rai?n"	"ran"、 "rain"
{ n }	匹配上一个元素恰好 n 次。	",\d{3}"	"1,043.6" 中的 ",043", "9,876,543,210" 中的 ",876"、 ",543" 和 ",210"
{ n ,}	匹配上一个元素至少 n 次。	"\d{2,}"	"166"、 "29"、 "1930"
{ n , m }	匹配上一个元素至少 n 次，但不多于 m 次。	"\d{3,5}"	"166"， "17668"， "193024" 中的 "19302"
*?	匹配上一个元素零次或多次，但次数尽可能少。	\d*?.\d	".0"、 "19.9"、 "219.9"
+?	匹配上一个元素一次或多次，但次数尽可能少。	"be+?"	"been" 中的 "be"， "bent" 中的 "be"
??	匹配上一个元素零次或一次，但次数尽可能少。	"rai??n"	"ran"、 "rain"
{ n }?	匹配前导元素恰好 n 次。	",\d{3}?"	"1,043.6" 中的 ",043", "9,876,543,210" 中的 ",876"、 ",543" 和 ",210"
{ n ,}?	匹配上一个元素至少 n 次，但次数尽可能少。	"\d{2,}?"	"166"、 "29" 和 "1930"
{ n , m }?	匹配上一个元素的次数介于 n 和 m 之间，但次数尽可能少。	"\d{3,5}?"	"166"， "17668"， "193024" 中的 "193" 和 "024"

反向引用构造

反向引用允许在同一正则表达式中随后标识以前匹配的子表达式。 下表列出了反向引用构造：

反向引用构造	描述	模式	匹配
\ number	反向引用。 匹配编号子表达式的值。	(\w)\1	"seek" 中的 "ee"
\k< name >	命名反向引用。 匹配命名表达式的值。	(?< char>\w)\k< char>	"seek" 中的 "ee"

备用构造

备用构造用于修改正则表达式以启用 either/or 匹配。 下表列出了备用构造：

<table>
<tr><th>备用构造</th><th>描述</th><th>模式</th><th>匹配</th></tr>
<tr><td>|</td><td>匹配以竖线 (|) 字符分隔的任何一个元素。</td><td>th(e|is|at)</td><td>"this is the day. " 中的 "the" 和 "this"</td></tr>
<tr><td>**(?(expression)yes | no)**</td><td>如果正则表达式模式由 expression 匹配指定，则匹配 *yes*；否则匹配可选的 *no* 部分。 expression 被解释为零宽度断言。</td><td>(?(A)A\d{2}\b|\b\d{3}\b)</td><td>"A10 C103 910" 中的 "A10" 和 "910"</td></tr>
<tr><td>**(?(name)yes | no)**</td><td>如果 name 或已命名或已编号的捕获组具有匹配，则匹配 *yes*；否则匹配可选的 *no*。</td><td>(?< quoted>")?(?(quoted).+?"|\S+\s)</td><td>"Dogs.jpg "Yiska playing.jpg"" 中的 Dogs.jpg 和 "Yiska playing.jpg"</td></tr>
</table>

替换

替换是替换模式中使用的正则表达式。 下表列出了用于替换的字符：

字符	描述	模式	替换模式	输入字符串	结果字符串
$number	替换按组 *number* 匹配的子字符串。	\b(\w+)(\s)(\w+)\b	$3$2$1	"one two"	"two one"
${name}	替换按命名组 *name* 匹配的子字符串。	\b(?<word1>\w+)(\s)(?<word2>\w+)\b	${word2} ${word1}	"one two"	"two one"
$	替换字符"$"。	\b(\d+)\s?USD	$1	"103 USD"	"$103"
><	替换整个匹配项的一个副本。	($(\d(.+\d+)?){1})	**><	"$1.30"	"**$1.30**"
**	替换匹配前的输入字符串的所有文本。	B+	"AABBCC"	"AAAACC"	
/tr>	替换匹配后的输入字符串的所有文本。	B+	/tr>	"AABBCC"	"AACCCC"
$+	替换最后捕获的组。	B+(C+)	$+	"AABBCCDD"	AACCDD
$_	替换整个输入字符串。	B+	$_	"AABBCC"	"AAAABBCCCC

杂项构造 下表列出了各种杂项构造：

构造	描述	实例
(?imnsx-imnsx)	在模式中间对诸如不区分大小写这样的选项进行设置或禁用。	\bA(?i)b\w+\b 匹配 "ABA Able Act" 中的 "ABA" 和 "Able"
(?#comment)	内联注释。该注释在第一个右括号处终止。	\bA(?#Matches words starting with A)\w+\b
# [to end of line]	X 模式注释。 该注释以非转义的 # 开头，并继续到行的结尾。	(?x)\bA\w+\b#Matches words starting with A

Regex 类

Regex 类用于表示一个正则表达式。 下表列出了 Regex 类中一些常用的方法：

方法	描述
public bool IsMatch(string input)	指示 Regex 构造函数中指定的正则表达式是否在指定的输入字符串中找到匹配项。
public bool IsMatch(string input, int startat)	指示 Regex 构造函数中指定的正则表达式是否在指定的输入字符串中找到匹配项，从字符串中指定的开始位置开始。
public static bool IsMatch(string input, string pattern)	指示指定的正则表达式是否在指定的输入字符串中找到匹配项。
public MatchCollection Matches(string input)	在指定的输入字符串中搜索正则表达式的所有匹配项。
public string Replace(string input, string replacement)	在指定的输入字符串中，把所有匹配正则表达式模式的所有匹配的字符串替换为指定的替换字符串。
public string[] Split(string input)	把输入字符串分割为子字符串数组，根据在 Regex 构造函数中指定的正则表达式模式定义的位置进行分割。

如需了解 Regex 类的完整的属性列表，请参阅微软的 C# 文档。

实例 1

下面的实例匹配了以 'S' 开头的单词：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "A Thousand Splendid Suns";

         Console.WriteLine("Matching words that start with 'S': ");
         showMatch(str, @"\bS\S*");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Matching words that start with 'S':
The Expression: \bS\S*
Splendid
Suns
```

实例 2

下面的实例匹配了以 'm' 开头以 'e' 结尾的单词：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "make maze and manage to measure it";

         Console.WriteLine("Matching words start with 'm' and ends with 'e':");
         showMatch(str, @"\bm\S*e\b");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
manage
measure
```

实例 3

下面的实例替换掉多余的空格：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         string input = "Hello   World   ";
         string pattern = "\\s+";
         string replacement = " ";
         Regex rgx = new Regex(pattern);
         string result = rgx.Replace(input, replacement);

         Console.WriteLine("Original String: {0}", input);
         Console.WriteLine("Replacement String: {0}", result);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Original String: Hello   World
Replacement String: Hello World
```

C# 异常处理

异常是在程序执行期间出现的问题。C# 中的异常是对程序运行时出现的特殊情况的一种响应，比如尝试除以零。

异常提供了一种把程序控制权从某个部分转移到另一个部分的方式。C# 异常处理时建立在四个关键词之上的：**try**、**catch**、**finally** 和 **throw**。

- **try**：一个 try 块标识了一个将被激活的特定的异常的代码块。后跟一个或多个 catch 块。
- **catch**：程序通过异常处理程序捕获异常。catch 关键字表示异常的捕获。
- **finally**：finally 块用于执行给定的语句，不管异常是否被抛出都会执行。例如，如果您打开一个文件，不管是否出现异常文件都要被关闭。
- **throw**：当问题出现时，程序抛出一个异常。使用 throw 关键字来完成。

语法

假设一个块将出现异常，一个方法使用 try 和 catch 关键字捕获异常。try/catch 块内的代码为受保护的代码，使用 try/catch 语法如下所示：

```
try
{
   // 引起异常的语句
}
catch( ExceptionName e1 )
{
   // 错误处理代码
}
catch( ExceptionName e2 )
{
   // 错误处理代码
}
catch( ExceptionName eN )
{
   // 错误处理代码
}
finally
{
   // 要执行的语句
}
```

您可以列出多个 catch 语句捕获不同类型的异常，以防 try 块在不同的情况下生成多个异常。

C# 中的异常类

C# 异常是使用类来表示的。C# 中的异常类主要是直接或间接地派生于 **System.Exception** 类。**System.ApplicationException** 和 **System.SystemException** 类是派生于 System.Exception 类的异常类。

System.ApplicationException 类支持由应用程序生成的异常。所以程序员定义的异常都应派生自该类。

System.SystemException 类是所有预定义的系统异常的基类。

下表列出了一些派生自 Sytem.SystemException 类的预定义的异常类：

异常类	描述
System.IO.IOException	处理 I/O 错误。
System.IndexOutOfRangeException	处理当方法指向超出范围的数组索引时生成的错误。
System.ArrayTypeMismatchException	处理当数组类型不匹配时生成的错误。
System.NullReferenceException	处理当依从一个空对象时生成的错误。
System.DivideByZeroException	处理当除以零时生成的错误。
System.InvalidCastException	处理在类型转换期间生成的错误。
System.OutOfMemoryException	处理空闲内存不足生成的错误。
System.StackOverflowException	处理栈溢出生成的错误。

异常处理

C# 以 try 和 catch 块的形式提供了一种结构化的异常处理方案。使用这些块，把核心程序语句与错误处理语句分离开。

这些错误处理块是使用 **try**、**catch** 和 **finally** 关键字实现的。下面是一个当除以零时抛出异常的实例：

```
using System;
namespace ErrorHandlingApplication
{
    class DivNumbers
    {
        int result;
        DivNumbers()
        {
            result = 0;
        }
        public void division(int num1, int num2)
        {
            try
            {
                result = num1 / num2;
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("Exception caught: {0}", e);
            }
            finally
            {
                Console.WriteLine("Result: {0}", result);
            }

        }
        static void Main(string[] args)
        {
            DivNumbers d = new DivNumbers();
            d.division(25, 0);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

创建用户自定义异常

您也可以定义自己的异常。用户自定义的异常类是派生自 **ApplicationException** 类。下面的实例演示了这点：

```
using System;
namespace UserDefinedException
{
   class TestTemperature
   {
      static void Main(string[] args)
      {
         Temperature temp = new Temperature();
         try
         {
            temp.showTemp();
         }
         catch(TempIsZeroException e)
         {
            Console.WriteLine("TempIsZeroException: {0}", e.Message);
         }
         Console.ReadKey();
      }
   }
}
public class TempIsZeroException: ApplicationException
{
   public TempIsZeroException(string message): base(message)
   {
   }
}
public class Temperature
{
   int temperature = 0;
   public void showTemp()
   {
      if(temperature == 0)
      {
         throw (new TempIsZeroException("Zero Temperature found"));
      }
      else
      {
         Console.WriteLine("Temperature: {0}", temperature);
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
TempIsZeroException: Zero Temperature found
```

抛出对象

如果异常是直接或间接派生自 **System.Exception** 类，您可以抛出一个对象。您可以在 catch 块中使用 throw 语句来抛出当前的对象，如下所示：

```
Catch(Exception e)
{
   ...
   Throw e
}
```

C# 文件的输入与输出

一个 文件 是一个存储在磁盘中带有指定名称和目录路径的数据集合。当打开文件进行读写时，它变成一个 流。

从根本上说，流是通过通信路径传递的字节序列。有两个主要的流：输入流 和 输出流。输入流用于从文件读取数据（读操作），输出流用于向文件写入数据（写操作）。

C# I/O 类

System.IO 命名空间有各种不同的类，用于执行各种文件操作，如创建和删除文件、读取或写入文件，关闭文件等。

下表列出了一些 System.IO 命名空间中常用的非抽象类：

I/O 类	描述
BinaryReader	从二进制流读取原始数据。
BinaryWriter	以二进制格式写入原始数据。
BufferedStream	字节流的临时存储。
Directory	有助于操作目录结构。
DirectoryInfo	用于对目录执行操作。
DriveInfo	提供驱动器的信息。
File	有助于处理文件。
FileInfo	用于对文件执行操作。
FileStream	用于文件中任何位置的读写。
MemoryStream	用于随机访问存储在内存中的数据流。
Path	对路径信息执行操作。
StreamReader	用于从字节流中读取字符。
StreamWriter	用于向一个流中写入字符。
StringReader	用于读取字符串缓冲区。
StringWriter	用于写入字符串缓冲区。

FileStream 类

System.IO 命名空间中的 **FileStream** 类有助于文件的读写与关闭。该类派生自抽象类 Stream。

您需要创建一个 **FileStream** 对象来创建一个新的文件，或打开一个已有的文件。创建 **FileStream** 对象的语法如下：

```
FileStream <object_name> = new FileStream( <file_name>,
<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);
```

例如，创建一个 FileStream 对象 **F** 来读取名为 **sample.txt** 的文件：

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Rea
```

参数	描述
FileMode	**FileMode** 枚举定义了各种打开文件的方法。FileMode 枚举的成员有：**Append**：打开一个已有的文件，并将光标放置在文件的末尾。如果文件不存在，则创建文件。 ***Create***：创建一个新的文件。如果文件已存在，则删除旧文件，然后创建新文件。 ***CreateNew***：指定操作系统应创建一个新的文件。如果文件已存在，则抛出异常。 ***Open***：打开一个已有的文件。如果文件不存在，则抛出异常。 ***OpenOrCreate***：指定操作系统应打开一个已有的文件。如果文件不存在，则用指定的名称创建一个新的文件打开。 **Truncate*：打开一个已有的文件，文件一旦打开，就将被截断为零字节大小。然后我们可以向文件写入全新的数据，但是保留文件的初始创建日期。如果文件不存在，则抛出异常。
FileAccess	**FileAccess** 枚举的成员有：**Read**、**ReadWrite** 和 **Write**。
FileShare	**FileShare** 枚举的成员有： **Inheritable**：允许文件句柄可由子进程继承。Win32 不直接支持此功能。 **None**：谢绝共享当前文件。文件关闭前，打开该文件的任何请求（由此进程或另一进程发出的请求）都将失败。 **Read**：允许随后打开文件读取。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取的请求（由此进程或另一进程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 **ReadWrite**：允许随后打开文件读取或写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取或写入的请求（由此进程或另一进程发出）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 **Write**：允许随后打开文件写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行写入的请求（由此进程或另一进过程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 **Delete**：允许随后删除文件。

实例

下面的程序演示了 **FileStream** 类的用法：

```
using System;
using System.IO;

namespace FileIOApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream F = new FileStream("test.dat",
            FileMode.OpenOrCreate, FileAccess.ReadWrite);

            for (int i = 1; i <= 20; i++)
            {
                F.WriteByte((byte)i);
            }

            F.Position = 0;

            for (int i = 0; i <= 20; i++)
            {
                Console.Write(F.ReadByte() + " ");
            }
            F.Close();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

C# 高级文件操作

上面的实例演示了 C# 中简单的文件操作。但是，要充分利用 C# System.IO 类的强大功能，您需要知道这些类常用的属性和方法。

在下面的章节中，我们将讨论这些类和它们执行的操作。请单击链接详细了解各个部分的知识：

主题	描述
文本文件的读写	它涉及到文本文件的读写。**StreamReader** 和 **StreamWriter** 类有助于完成文本文件的读写。
二进制文件的读写	它涉及到二进制文件的读写。**BinaryReader** 和 **BinaryWriter** 类有助于完成二进制文件的读写。
Windows 文件系统的操作	它让 C# 程序员能够浏览并定位 Windows 文件和目录。

C# 高级

C# 特性（Attribute）

特性（**Attribute**）是用于在运行时传递程序中各种元素（比如类、方法、结构、枚举、组件等）的行为信息的声明性标签。您可以通过使用特性向程序添加声明性信息。一个声明性标签是通过放置在它所应用的元素前面的方括号（[ ]）来描述的。

特性（Attribute）用于添加元数据，如编译器指令和注释、描述、方法、类等其他信息。.Net 框架提供了两种类型的特性：预定义特性和自定义特性。

规定特性（Attribute）

规定特性（Attribute）的语法如下：

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

特性（Attribute）的名称和值是在方括号内规定的，放置在它所应用的元素之前。positional_parameters 规定必需的信息，name_parameter 规定可选的信息。

预定义特性（Attribute）

.Net 框架提供了三种预定义特性：

- AttributeUsage
- Conditional
- Obsolete

AttributeUsage

预定义特性 **AttributeUsage** 描述了如何使用一个自定义特性类。它规定了特性可应用到的项目的类型。

规定该特性的语法如下：

```
[AttributeUsage(
   validon,
   AllowMultiple=allowmultiple,
   Inherited=inherited
)]
```

其中：

- 参数 validon 规定特性可被放置的语言元素。它是枚举器 *AttributeTargets* 的值的组合。默认值是 *AttributeTargets.All*。
- 参数 *allowmultiple*（可选的）为该特性的 *AllowMultiple* 属性（property）提供一个布尔值。如果为 true，则该特性是多用的。默认值是 false（单用的）。
- 参数 *inherited*（可选的）为该特性的 *Inherited* 属性（property）提供一个布尔值。如果为 true，则该特性可被派生类继承。默认值是 false（不被继承）。

例如：

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Feild |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
```

Conditional

这个预定义特性标记了一个条件方法，其执行依赖于它顶的预处理标识符。

它会引起方法调用的条件编译，取决于指定的值，比如 **Debug** 或 **Trace**。例如，当调试代码时显示变量的值。

规定该特性的语法如下：

```
[Conditional(
   conditionalSymbol
)]
```

例如：

```
[Conditional("DEBUG")]
```

下面的实例演示了该特性：

```
#define DEBUG
using System;
using System.Diagnostics;
public class Myclass
{
    [Conditional("DEBUG")]
    public static void Message(string msg)
    {
        Console.WriteLine(msg);
    }
}
class Test
{
    static void function1()
    {
        Myclass.Message("In Function 1.");
        function2();
    }
    static void function2()
    {
        Myclass.Message("In Function 2.");
    }
    public static void Main()
    {
        Myclass.Message("In Main function.");
        function1();
        Console.ReadKey();
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main function
In Function 1
In Function 2
```

Obsolete

这个预定义特性标记了不应被使用的程序实体。它可以让您通知编译器丢弃某个特定的目标元素。例如，当一个新方法被用在一个类中，但是您仍然想要保持类中的旧方法，您可以通过显示一个应该使用新方法，而不是旧方法的消息，来把它标记为 obsolete（过时的）。

规定该特性的语法如下：

```
[Obsolete(
   message
)]
[Obsolete(
   message,
   iserror
)]
```

其中：

- 参数 *message*，是一个字符串，描述项目为什么过时的原因以及该替代使用什么。
- 参数 *iserror*，是一个布尔值。如果该值为 true，编译器应把该项目的使用当作一个错误。默认值是 false（编译器生成一个警告）。

下面的实例演示了该特性：

```
using System;
public class MyClass
{
   [Obsolete("Don't use OldMethod, use NewMethod instead", true)]
   static void OldMethod()
   {
      Console.WriteLine("It is the old method");
   }
   static void NewMethod()
   {
      Console.WriteLine("It is the new method");
   }
   public static void Main()
   {
      OldMethod();
   }
}
```

当您尝试编译该程序时，编译器会给出一个错误消息说明：

```
Don't use OldMethod, use NewMethod instead
```

创建自定义特性（Attribute）

.Net 框架允许创建自定义特性，用于存储声明性的信息，且可在运行时被检索。该信息根据设计标准和应用程序需要，可与任何目标元素相关。

创建并使用自定义特性包含四个步骤：

- 声明自定义特性
- 构建自定义特性
- 在目标程序元素上应用自定义特性
- 通过反射访问特性

最后一个步骤包含编写一个简单的程序来读取元数据以便查找各种符号。元数据是用于描述其他数据的数据和信息。该程序应使用反射来在运行时访问特性。我们将在下一章详细讨论这点。

声明自定义特性

一个新的自定义特性应派生自 **System.Attribute** 类。例如：

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

在上面的代码中，我们已经声明了一个名为 *DeBugInfo* 的自定义特性。

构建自定义特性

让我们构建一个名为 *DeBugInfo* 的自定义特性，该特性将存储调试程序获得的信息。它存储下面的信息：

- bug 的代码编号
- 辨认该 bug 的开发人员名字
- 最后一次审查该代码的日期
- 一个存储了开发人员标记的字符串消息

我们的 *DeBugInfo* 类将带有三个用于存储前三个信息的私有属性（property）和一个用于存储消息的公有属性（property）。所以 bug 编号、开发人员名字和审查日期将是 DeBugInfo 类的必需的定位（ positional）参数，消息将是一个可选的命名（named）参数。

每个特性必须至少有一个构造函数。必需的定位（ positional）参数应通过构造函数传递。下面的代码演示了 *DeBugInfo* 类：

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
  private int bugNo;
  private string developer;
  private string lastReview;
  public string message;

  public DeBugInfo(int bg, string dev, string d)
  {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
  }

  public int BugNo
  {
      get
      {
          return bugNo;
      }
  }
  public string Developer
  {
      get
      {
          return developer;
      }
  }
  public string LastReview
  {
      get
      {
          return lastReview;
      }
  }
  public string Message
  {
      get
      {
          return message;
      }
      set
      {
          message = value;
      }
  }
}
```

应用自定义特性

通过把特性放置在紧接着它的目标之前，来应用该特性：

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle
{
  // 成员变量
  protected double length;
  protected double width;
  public Rectangle(double l, double w)
  {
      length = l;
      width = w;
  }
  [DeBugInfo(55, "Zara Ali", "19/10/2012",
  Message = "Return type mismatch")]
  public double GetArea()
  {
      return length * width;
  }
  [DeBugInfo(56, "Zara Ali", "19/10/2012")]
  public void Display()
  {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
  }
}
```

在下一章中，我们将使用 Reflection 类对象来检索这些信息。

C# 反射（Reflection）

反射（**Reflection**） 对象用于在运行时获取类型信息。该类位于 **System.Reflection** 命名空间中，可访问一个正在运行的程序的元数据。

System.Reflection 命名空间包含了允许您获取有关应用程序信息及向应用程序动态添加类型、值和对象的类。

反射（Reflection）的用途

反射（Reflection）有下列用途：

- 它允许在运行时查看属性（attribute）信息。
- 它允许审查集合中的各种类型，以及实例化这些类型。
- 它允许延迟绑定的方法和属性（property）。
- 它允许在运行时创建新类型，然后使用这些类型执行一些任务。

查看元数据

我们已经在上面的章节中提到过，使用反射（Reflection）可以查看属性（attribute）信息。

System.Reflection 类的 **MemberInfo** 对象需要被初始化，用于发现与类相关的属性（attribute）。为了做到这点，您可以定义目标类的一个对象，如下：

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

下面的程序演示了这点：

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute
{
   public readonly string Url;

   public string Topic  // Topic 是一个命名（named）参数
   {
      get
      {
         return topic;
      }
      set
      {

         topic = value;
      }
   }

   public HelpAttribute(string url)  // url 是一个定位（positional）参数
   {
      this.Url = url;
   }

   private string topic;
}
[HelpAttribute("Information on the class MyClass")]
class MyClass
{
}

namespace AttributeAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         System.Reflection.MemberInfo info = typeof(MyClass);
         object[] attributes = info.GetCustomAttributes(true);
         for (int i = 0; i < attributes.Length; i++)
         {
            System.Console.WriteLine(attributes[i]);
         }
         Console.ReadKey();

      }
   }
}
```

当上面的代码被编译和执行时，它会显示附加到类 *MyClass* 上的自定义属性：

```
HelpAttribute
```

实例

在本实例中，我们将使用在上一章中创建的 *DeBugInfo* 属性，并使用反射（Reflection）来读取 *Rectangle* 类中的元数据。

```
using System;
using System.Reflection;
```

```
namespace BugFixApplication
{
   // 一个自定义属性 BugFix 被赋给类及其成员
   [AttributeUsage(AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]

   public class DeBugInfo : System.Attribute
   {
      private int bugNo;
      private string developer;
      private string lastReview;
      public string message;

      public DeBugInfo(int bg, string dev, string d)
      {
         this.bugNo = bg;
         this.developer = dev;
         this.lastReview = d;
      }

      public int BugNo
      {
         get
         {
            return bugNo;
         }
      }
      public string Developer
      {
         get
         {
            return developer;
         }
      }
      public string LastReview
      {
         get
         {
            return lastReview;
         }
      }
      public string Message
      {
         get
         {
            return message;
         }
         set
         {
            message = value;
         }
      }
   }
   [DeBugInfo(45, "Zara Ali", "12/8/2012",
    Message = "Return type mismatch")]
   [DeBugInfo(49, "Nuha Ali", "10/10/2012",
    Message = "Unused variable")]
   class Rectangle
   {
      // 成员变量
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }
      [DeBugInfo(55, "Zara Ali", "19/10/2012",
```

```
         Message = "Return type mismatch")]
        public double GetArea()
        {
           return length * width;
        }
        [DeBugInfo(56, "Zara Ali", "19/10/2012")]
        public void Display()
        {
           Console.WriteLine("Length: {0}", length);
           Console.WriteLine("Width: {0}", width);
           Console.WriteLine("Area: {0}", GetArea());
        }
     }//end class Rectangle

     class ExecuteRectangle
     {
        static void Main(string[] args)
        {
           Rectangle r = new Rectangle(4.5, 7.5);
           r.Display();
           Type type = typeof(Rectangle);
           // 遍历 Rectangle 类的属性
           foreach (Object attributes in type.GetCustomAttributes(false))
           {
              DeBugInfo dbi = (DeBugInfo)attributes;
              if (null != dbi)
              {
                 Console.WriteLine("Bug no: {0}", dbi.BugNo);
                 Console.WriteLine("Developer: {0}", dbi.Developer);
                 Console.WriteLine("Last Reviewed: {0}",
                      dbi.LastReview);
                 Console.WriteLine("Remarks: {0}", dbi.Message);
              }
           }

           // 遍历方法属性
           foreach (MethodInfo m in type.GetMethods())
           {
              foreach (Attribute a in m.GetCustomAttributes(true))
              {
                 DeBugInfo dbi = (DeBugInfo)a;
                 if (null != dbi)
                 {
                    Console.WriteLine("Bug no: {0}, for Method: {1}",
                          dbi.BugNo, m.Name);
                    Console.WriteLine("Developer: {0}", dbi.Developer);
                    Console.WriteLine("Last Reviewed: {0}",
                          dbi.LastReview);
                    Console.WriteLine("Remarks: {0}", dbi.Message);
                 }
              }
           }
           Console.ReadLine();
        }
     }
  }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Length: 4.5
Width: 7.5
Area: 33.75
Bug No: 49
Developer: Nuha Ali
Last Reviewed: 10/10/2012
Remarks: Unused variable
Bug No: 45
Developer: Zara Ali
Last Reviewed: 12/8/2012
Remarks: Return type mismatch
Bug No: 55, for Method: GetArea
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks: Return type mismatch
Bug No: 56, for Method: Display
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks:
```

C# 属性（Property）

属性（**Property**） 是类（class）、结构（structure）和接口（interface）的命名（named）成员。类或结构中的成员变量或方法称为 域（**Field**）。属性（Property）是域（Field）的扩展，且可使用相同的语法来访问。它们使用 访问器（**accessors**） 让私有域的值可被读写或操作。

属性（Property）不会确定存储位置。相反，它们具有可读写或计算它们值的 访问器（**accessors**）。

例如，有一个名为 Student 的类，带有 age、name 和 code 的私有域。我们不能在类的范围以外直接访问这些域，但是我们可以拥有访问这些私有域的属性。

访问器（Accessors）

属性（Property）的访问器（**accessor**）包含有助于获取（读取或计算）或设置（写入）属性的可执行语句。访问器（accessor）声明可包含一个 get 访问器、一个 set 访问器，或者同时包含二者。例如：

```
// 声明类型为 string 的 Code 属性
public string Code
{
   get
   {
      return code;
   }
   set
   {
      code = value;
   }
}

// 声明类型为 string 的 Name 属性
public string Name
{
   get
   {
     return name;
   }
   set
   {
     name = value;
   }
}

// 声明类型为 int 的 Age 属性
public int Age
{
   get
   {
      return age;
   }
   set
   {
      age = value;
   }
}
```

实例

下面的实例演示了属性（Property）的用法：

```
using System;
namespace tutorialspoint
{
   class Student
   {

      private string code = "N.A";
      private string name = "not known";
      private int age = 0;

      // 声明类型为 string 的 Code 属性
      public string Code
      {
         get
         {
            return code;
         }
         set
         {
            code = value;
         }
      }
```

```
        // 声明类型为 string 的 Name 属性
        public string Name
        {
           get
           {
              return name;
           }
           set
           {
              name = value;
           }
        }

        // 声明类型为 int 的 Age 属性
        public int Age
        {
           get
           {
              return age;
           }
           set
           {
              age = value;
           }
        }
        public override string ToString()
        {
           return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
        }
      }
      class ExampleDemo
      {
        public static void Main()
        {
           // 创建一个新的 Student 对象
           Student s = new Student();

           // 设置 student 的 code、name 和 age
           s.Code = "001";
           s.Name = "Zara";
           s.Age = 9;
           Console.WriteLine("Student Info: {0}", s);
           // 增加年龄
           s.Age += 1;
           Console.WriteLine("Student Info: {0}", s);
           Console.ReadKey();
         }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

抽象属性（Abstract Properties）

抽象类可拥有抽象属性，这些属性应在派生类中被实现。下面的程序说明了这点：

```
using System;
namespace tutorialspoint
{
```

```
    public abstract class Person
    {
       public abstract string Name
       {
          get;
          set;
       }
       public abstract int Age
       {
          get;
          set;
       }
    }
    class Student : Person
    {

       private string code = "N.A";
       private string name = "N.A";
       private int age = 0;

       // 声明类型为 string 的 Code 属性
       public string Code
       {
          get
          {
             return code;
          }
          set
          {
             code = value;
          }
       }

       // 声明类型为 string 的 Name 属性
       public override string Name
       {
          get
          {
             return name;
          }
          set
          {
             name = value;
          }
       }

       // 声明类型为 int 的 Age 属性
       public override int Age
       {
          get
          {
             return age;
          }
          set
          {
             age = value;
          }
       }
       public override string ToString()
       {
          return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
       }
    }
    class ExampleDemo
    {
       public static void Main()
       {
          // 创建一个新的 Student 对象
          Student s = new Student();

          // 设置 student 的 code、name 和 age
          s.Code = "001";
```

```
            s.Name = "Zara";
            s.Age = 9;
            Console.WriteLine("Student Info:- {0}", s);
            // 增加年龄
            s.Age += 1;
            Console.WriteLine("Student Info:- {0}", s);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

C# 索引器（Indexer）

索引器（**Indexer**） 允许一个对象可以像数组一样被索引。当您为类定义一个索引器时，该类的行为就会像一个 虚拟数组（**virtual array**） 一样。您可以使用数组访问运算符（[ ]）来访问该类的实例。

语法

一维索引器的语法如下：

```
element-type this[int index]
{
   // get 访问器
   get
   {
      // 返回 index 指定的值
   }

   // set 访问器
   set
   {
      // 设置 index 指定的值
   }
}
```

索引器（Indexer）的用途

索引器的行为的声明在某种程度上类似于属性（property）。就像属性（property），您可使用 **get** 和 **set** 访问器来定义索引器。但是，属性返回或设置一个特定的数据成员，而索引器返回或设置对象实例的一个特定值。换句话说，它把实例数据分为更小的部分，并索引每个部分，获取或设置每个部分。

定义一个属性（property）包括提供属性名称。索引器定义的时候不带有名称，但带有 **this** 关键字，它指向对象实例。下面的实例演示了这个概念：

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
      {
         for (int i = 0; i < size; i++)
         namelist[i] = "N. A.";
      }
      public string this[int index]
      {
         get
         {
            string tmp;

            if( index >= 0 && index <= size-1 )
            {
               tmp = namelist[index];
            }
            else
            {
               tmp = "";
            }

            return ( tmp );
         }
         set
         {
            if( index >= 0 && index <= size-1 )
            {
               namelist[index] = value;
            }
         }
      }

      static void Main(string[] args)
      {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";
         for ( int i = 0; i < IndexedNames.size; i++ )
         {
            Console.WriteLine(names[i]);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
```

重载索引器（Indexer）

索引器（Indexer）可被重载。索引器声明的时候也可带有多个参数，且每个参数可以是不同的类型。没有必要让索引器必须是整型的。C# 允许索引器可以是其他类型，例如，字符串类型。

下面的实例演示了重载索引器：

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
      {
         for (int i = 0; i < size; i++)
         {
          namelist[i] = "N. A.";
         }
      }
      public string this[int index]
      {
         get
         {
            string tmp;

            if( index >= 0 && index <= size-1 )
            {
               tmp = namelist[index];
            }
            else
            {
               tmp = "";
            }

            return ( tmp );
         }
         set
         {
            if( index >= 0 && index <= size-1 )
            {
               namelist[index] = value;
            }
         }
      }
      public int this[string name]
      {
         get
         {
            int index = 0;
```

```
            while(index < size)
            {
               if (namelist[index] == name)
               {
                return index;
               }
               index++;
            }
            return index;
         }

      }

      static void Main(string[] args)
      {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";
         // 使用带有 int 参数的第一个索引器
         for (int i = 0; i < IndexedNames.size; i++)
         {
            Console.WriteLine(names[i]);
         }
         // 使用带有 string 参数的第二个索引器
         Console.WriteLine(names["Nuha"]);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2
```

C# 委托（Delegate）

C# 中的委托（Delegate）类似于 C 或 C++ 中函数的指针。委托（**Delegate**） 是存有对某个方法的引用的一种引用类型变量。引用可在运行时被改变。

委托（Delegate）特别用于实现事件和回调方法。所有的委托（Delegate）都派生自 **System.Delegate** 类。

声明委托（Delegate）

委托声明决定了可由该委托引用的方法。委托可指向一个与其具有相同标签的方法。

例如，假设有一个委托：

```
public delegate int MyDelegate (string s);
```

上面的委托可被用于引用任何一个带有一个单一的 *string* 参数的方法，并返回一个 *int* 类型变量。

声明委托的语法如下：

```
delegate <return type> <delegate-name> <parameter list>
```

实例化委托（Delegate）

一旦声明了委托类型，委托对象必须使用 **new** 关键字来创建，且与一个特定的方法有关。当创建委托时，传递到 **new** 语句的参数就像方法调用一样书写，但是不带有参数。例如：

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

下面的实例演示了委托的声明、实例化和使用，该委托可用于引用带有一个整型参数的方法，并返回一个整型值。

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
      public static int AddNum(int p)
      {
         num += p;
         return num;
      }

      public static int MultNum(int q)
      {
         num *= q;
         return num;
      }
      public static int getNum()
      {
         return num;
      }

      static void Main(string[] args)
      {
         // 创建委托实例
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         // 使用委托对象调用方法
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());
         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of Num: 35
Value of Num: 175
```

委托的多播（Multicasting of a Delegate）

委托对象可使用 "+" 运算符进行合并。一个合并委托调用它所合并的两个委托。只有相同类型的委托可被合并。"-" 运算符可用于从合并的委托中移除组件委托。

使用委托的这个有用的特点，您可以创建一个委托被调用时要调用的方法的调用列表。这被称为委托的 多播（**multicasting**），也叫组播。下面的程序演示了委托的多播：

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
      public static int AddNum(int p)
      {
         num += p;
         return num;
      }

      public static int MultNum(int q)
      {
         num *= q;
         return num;
      }
      public static int getNum()
      {
         return num;
      }

      static void Main(string[] args)
      {
         // 创建委托实例
         NumberChanger nc;
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         nc = nc1;
         nc += nc2;
         // 调用多播
         nc(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of Num: 75
```

委托（Delegate）的用途

下面的实例演示了委托的用法。委托 *printString* 可用于引用带有一个字符串作为输入的方法，并不返回任何东西。

我们使用这个委托来调用两个方法，第一个把字符串打印到控制台，第二个把字符串打印到文件：

```
using System;
using System.IO;

namespace DelegateAppl
{
   class PrintString
   {
      static FileStream fs;
      static StreamWriter sw;
      // 委托声明
      public delegate void printString(string s);

      // 该方法打印到控制台
      public static void WriteToScreen(string str)
      {
         Console.WriteLine("The String is: {0}", str);
      }
      // 该方法打印到文件
      public static void WriteToFile(string s)
      {
         fs = new FileStream("c:\\message.txt",
         FileMode.Append, FileAccess.Write);
         sw = new StreamWriter(fs);
         sw.WriteLine(s);
         sw.Flush();
         sw.Close();
         fs.Close();
      }
      // 该方法把委托作为参数，并使用它调用方法
      public static void sendString(printString ps)
      {
         ps("Hello World");
      }
      static void Main(string[] args)
      {
         printString ps1 = new printString(WriteToScreen);
         printString ps2 = new printString(WriteToFile);
         sendString(ps1);
         sendString(ps2);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The String is: Hello World
```

C# 事件（Event）

事件（**Event**） 基本上说是一个用户操作，如按键、点击、鼠标移动等等，或者是一些出现，如系统生成的通知。应用程序需要在事件发生时响应事件。例如，中断。事件是用于进程间通信。

通过事件使用委托

事件在类中声明且生成，且通过使用同一个类或其他类中的委托与事件处理程序关联。包含事件的类用于发布事件。这被称为 发布器（**publisher**） 类。其他接受该事件的类被称为 订阅器（**subscriber**） 类。事件使用 发布**-**订阅（**publisher-subscriber**） 模型。

发布器（**publisher**） 是一个包含事件和委托定义的对象。事件和委托之间的联系也定义在这个对象中。发布器（publisher）类的对象调用这个事件，并通知其他的对象。

订阅器（**subscriber**） 是一个接受事件并提供事件处理程序的对象。在发布器（publisher）类中的委托调用订阅器（subscriber）类中的方法（事件处理程序）。

声明事件（Event）

在类的内部声明事件，首先必须声明该事件的委托类型。例如：

```
public delegate void BoilerLogHandler(string status);
```

然后，声明事件本身，使用 **event** 关键字：

```
// 基于上面的委托定义事件
public event BoilerLogHandler BoilerEventLog;
```

上面的代码定义了一个名为 *BoilerLogHandler* 的委托和一个名为 *BoilerEventLog* 的事件，该事件在生成的时候会调用委托。

实例 1

```
using System;
namespace SimpleEvent
{
   using System;

   public class EventTest
   {
      private int value;

      public delegate void NumManipulationHandler();

      public event NumManipulationHandler ChangeNum;

      protected virtual void OnNumChanged()
      {
         if (ChangeNum != null)
         {
            ChangeNum();
         }
         else
         {
            Console.WriteLine("Event fired!");
         }

      }
      public EventTest(int n )
      {
         SetValue(n);
      }
      public void SetValue(int n)
      {
         if (value != n)
         {
            value = n;
            OnNumChanged();
         }
      }
   }
   public class MainClass
   {
      public static void Main()
      {
         EventTest e = new EventTest(5);
         e.SetValue(7);
         e.SetValue(11);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Event Fired!
Event Fired!
Event Fired!
```

实例 2

本实例提供一个简单的用于热水锅炉系统故障排除的应用程序。当维修工程师检查锅炉时，锅炉的温度和压力会随着维修工程师的备注自动记录到日志文件中。

```
using System;
using System.IO;

namespace BoilerEventAppl
{

   // boiler 类
   class Boiler
   {
      private int temp;
      private int pressure;
      public Boiler(int t, int p)
      {
         temp = t;
         pressure = p;
      }

      public int getTemp()
      {
         return temp;
      }
      public int getPressure()
      {
         return pressure;
      }
   }
   // 事件发布器
   class DelegateBoilerEvent
   {
      public delegate void BoilerLogHandler(string status);

      // 基于上面的委托定义事件
      public event BoilerLogHandler BoilerEventLog;

      public void LogProcess()
      {
         string remarks = "O. K";
         Boiler b = new Boiler(100, 12);
         int t = b.getTemp();
         int p = b.getPressure();
         if(t > 150 || t < 80 || p < 12 || p > 15)
         {
            remarks = "Need Maintenance";
         }
         OnBoilerEventLog("Logging Info:\n");
         OnBoilerEventLog("Temparature " + t + "\nPressure: " + p);
         OnBoilerEventLog("\nMessage: " + remarks);
      }

      protected void OnBoilerEventLog(string message)
      {
         if (BoilerEventLog != null)
         {
            BoilerEventLog(message);
         }
      }
   }
   // 该类保留写入日志文件的条款
   class BoilerInfoLogger
   {
      FileStream fs;
      StreamWriter sw;
      public BoilerInfoLogger(string filename)
      {
         fs = new FileStream(filename, FileMode.Append, FileAccess.Write);
         sw = new StreamWriter(fs);
      }
      public void Logger(string info)
      {
         sw.WriteLine(info);
      }
```

```
        public void Close()
        {
            sw.Close();
            fs.Close();
        }
    }
    // 事件订阅器
    public class RecordBoilerInfo
    {
        static void Logger(string info)
        {
            Console.WriteLine(info);
        }//end of Logger

        static void Main(string[] args)
        {
            BoilerInfoLogger filelog = new BoilerInfoLogger("e:\\boiler.txt");
            DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();
            boilerEvent.BoilerEventLog += new
            DelegateBoilerEvent.BoilerLogHandler(Logger);
            boilerEvent.BoilerEventLog += new
            DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);
            boilerEvent.LogProcess();
            Console.ReadLine();
            filelog.Close();
        }//end of main

    }//end of RecordBoilerInfo
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Logging info:

Temperature 100
Pressure 12

Message: O. K
```

C# 集合（Collection）

集合（Collection）类是专门用于数据存储和检索的类。这些类提供了对栈（stack）、队列（queue）、列表（list）和哈希表（hash table）的支持。大多数集合类实现了相同的接口。

集合（Collection）类服务于不同的目的，如为元素动态分配内存，基于索引访问列表项等等。这些类创建 Object 类的对象的集合。在 C# 中，Object 类是所有数据类型的基类。

各种集合类和它们的用法

下面是各种常用的 **System.Collection** 命名空间的类。点击下面的链接查看细节。

类	描述和用法
动态数组（ArrayList）	它代表了可被单独索引的对象的有序集合。 它基本上可以替代一个数组。但是，与数组不同的是，您可以使用索引在指定的位置添加和移除项目，动态数组会自动重新调整它的大小。它也允许在列表中进行动态内存分配、增加、搜索、排序各项。
哈希表（Hashtable）	它使用键来访问集合中的元素。 当您使用键访问元素时，则使用哈希表，而且您可以识别一个有用的键值。哈希表中的每一项都有一个键/值对。键用于访问集合中的项目。
排序列表（SortedList）	它可以使用键和索引来访问列表中的项。 排序列表是数组和哈希表的组合。它包含一个可使用键或索引访问各项的列表。如果您使用索引访问各项，则它是一个动态数组（ArrayList），如果您使用键访问各项，则它是一个哈希表（Hashtable）。集合中的各项总是按键值排序。
堆栈（Stack）	它代表了一个后进先出的对象集合。 当您需要对各项进行后进先出的访问时，则使用堆栈。当您在列表中添加一项，称为推入元素，当您从列表中移除一项时，称为弹出元素。
队列（Queue）	它代表了一个先进先出的对象集合。 当您需要对各项进行先进先出的访问时，则使用队列。当您在列表中添加一项，称为入队，当您从列表中移除一项时，称为出队。
点阵列（BitArray）	它代表了一个使用值 1 和 0 来表示的二进制数组。 当您需要存储位，但是事先不知道位数时，则使用点阵列。您可以使用整型索引从点阵列集合中访问各项，索引从零开始。

C# 泛型（Generic）

泛型（**Generic**） 允许您延迟编写类或方法中的编程元素的数据类型的规范，直到实际在程序中使用它的时候。换句话说，泛型允许您编写一个可以与任何数据类型一起工作的类或方法。

您可以通过数据类型的替代参数编写类或方法的规范。当编译器遇到类的构造函数或方法的函数调用时，它会生成代码来处理指定的数据类型。下面这个简单的实例将有助于您理解这个概念：

```
using System;
using System.Collections.Generic;

namespace GenericApplication
{
    public class MyGenericArray<T>
    {
        private T[] array;
        public MyGenericArray(int size)
        {
            array = new T[size + 1];
        }
        public T getItem(int index)
        {
            return array[index];
        }
        public void setItem(int index, T value)
        {
            array[index] = value;
        }
    }

    class Tester
    {
        static void Main(string[] args)
        {
            // 声明一个整型数组
            MyGenericArray<int> intArray = new MyGenericArray<int>(5);
            // 设置值
            for (int c = 0; c < 5; c++)
            {
                intArray.setItem(c, c*5);
            }
            // 获取值
            for (int c = 0; c < 5; c++)
            {
                Console.Write(intArray.getItem(c) + " ");
            }
            Console.WriteLine();
            // 声明一个字符数组
            MyGenericArray<char> charArray = new MyGenericArray<char>(5);
            // 设置值
            for (int c = 0; c < 5; c++)
            {
                charArray.setItem(c, (char)(c+97));
            }
            // 获取值
            for (int c = 0; c < 5; c++)
            {
                Console.Write(charArray.getItem(c) + " ");
            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
0 5 10 15 20
a b c d e
```

泛型（Generic）的特性

使用泛型是一种增强程序功能的技术，具体表现在以下几个方面：

- 它有助于您最大限度地重用代码、保护类型的安全以及提高性能。
- 您可以创建泛型集合类。.NET 框架类库在 *System.Collections.Generic* 命名空间中包含了一些新的泛型集合类。您可以使用这些泛型集合类来替代 *System.Collections* 中的集合类。
- 您可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。
- 您可以对泛型类进行约束以访问特定数据类型的方法。
- 关于泛型数据类型中使用的类型的信息可在运行时通过使用反射获取。

泛型（Generic）方法

在上面的实例中，我们已经使用了泛型类，我们可以通过类型参数声明泛型方法。下面的程序说明了这个概念：

```
using System;
using System.Collections.Generic;

namespace GenericMethodAppl
{
    class Program
    {
        static void Swap<T>(ref T lhs, ref T rhs)
        {
            T temp;
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }
        static void Main(string[] args)
        {
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'I';
            d = 'V';

            // 在交换之前显示值
            Console.WriteLine("Int values before calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values before calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            // 调用 swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            // 在交换之后显示值
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

泛型（Generic）委托

您可以通过类型参数定义泛型委托。例如：

```
delegate T NumberChanger<T>(T n);
```

下面的实例演示了委托的使用：

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl
{
    class TestDelegate
    {
        static int num = 10;
        public static int AddNum(int p)
        {
            num += p;
            return num;
        }

        public static int MultNum(int q)
        {
            num *= q;
            return num;
        }
        public static int getNum()
        {
            return num;
        }

        static void Main(string[] args)
        {
            // 创建委托实例
            NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
            NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);
            // 使用委托对象调用方法
            nc1(25);
            Console.WriteLine("Value of Num: {0}", getNum());
            nc2(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of Num: 35
Value of Num: 175
```

C# 匿名方法

我们已经提到过，委托是用于引用与其具有相同标签的方法。换句话说，您可以使用委托对象调用可由委托引用的方法。

匿名方法（**Anonymous methods**） 提供了一种传递代码块作为委托参数的技术。匿名方法是没有名称只有主体的方法。

在匿名方法中您不需要指定返回类型，它是从方法主体内的 return 语句推断的。

编写匿名方法的语法

匿名方法是通过使用 **delegate** 关键字创建委托实例来声明的。例如：

```
delegate void NumberChanger(int n);
...
NumberChanger nc = delegate(int x)
{
    Console.WriteLine("Anonymous Method: {0}", x);
};
```

代码块 *Console.WriteLine("Anonymous Method: {0}", x);* 是匿名方法的主体。

委托可以通过匿名方法调用，也可以通过命名方法调用，即，通过向委托对象传递方法参数。

例如：

```
nc(10);
```

实例

下面的实例演示了匿名方法的概念：

```
using System;

delegate void NumberChanger(int n);
namespace DelegateAppl
{
    class TestDelegate
    {
        static int num = 10;
        public static void AddNum(int p)
        {
            num += p;
            Console.WriteLine("Named Method: {0}", num);
        }

        public static void MultNum(int q)
        {
            num *= q;
            Console.WriteLine("Named Method: {0}", num);
        }
        public static int getNum()
        {
            return num;
        }

        static void Main(string[] args)
        {
            // 使用匿名方法创建委托实例
            NumberChanger nc = delegate(int x)
            {
               Console.WriteLine("Anonymous Method: {0}", x);
            };

            // 使用匿名方法调用委托
            nc(10);

            // 使用命名方法实例化委托
            nc =  new NumberChanger(AddNum);

            // 使用命名方法调用委托
            nc(5);

            // 使用另一个命名方法实例化委托
            nc =  new NumberChanger(MultNum);

            // 使用命名方法调用委托
            nc(2);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Anonymous Method: 10
Named Method: 15
Named Method: 30
```

C# 不安全代码

当一个代码块使用 **unsafe** 修饰符标记时，C# 允许在函数中使用指针变量。不安全代码或非托管代码是指使用了指针变量的代码块。

指针变量

指针 是值为另一个变量的地址的变量，即，内存位置的直接地址。就像其他变量或常量，您必须在使用指针存储其他变量地址之前声明指针。

指针变量声明的一般形式为：

```
type *var-name;
```

以下是有效的指针声明：

```
int    *ip;    /* 指向一个整数 */
double *dp;    /* 指向一个双精度数 */
float  *fp;    /* 指向一个浮点数 */
char   *ch     /* 指向一个字符 */
```

下面的实例说明了 C# 中使用了 **unsafe** 修饰符时指针的使用：

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ",  var);
            Console.WriteLine("Address is: {0}",  (int)p);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Data is: 20
Address is: 99215364
```

您也可以不用声明整个方法作为不安全代码，只需要声明方法的一部分作为不安全代码。下面的实例说明了这点。

使用指针检索数据值

您可以使用 **ToString()** 方法检索存储在指针变量所引用位置的数据。下面的实例演示了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Data is: 20
Data is: 20
Address is: 77128984
```

传递指针作为方法的参数

您可以向方法传递指针变量作为方法的参数。下面的实例说明了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

使用指针访问数组元素

在 C# 中，数组名称和一个指向与数组数据具有相同数据类型的指针是不同的变量类型。例如，int *p 和 int[] p 是不同的类型。您可以增加指针变量 p，因为它在内存中不是固定的，但是数组地址在内存中是固定的，所以您不能增加数组 p。

因此，如果您需要使用指针变量访问数组数据，可以像我们通常在 C 或 C++ 中所做的那样，使用 **fixed** 关键字来固定指针。

下面的实例演示了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* 显示指针中数组地址 */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

编译不安全代码

为了编译不安全代码，您必须切换到命令行编译器指定 **/unsafe** 命令行。

例如，为了编译包含不安全代码的名为 prog1.cs 的程序，需在命令行中输入命令：

```
csc /unsafe prog1.cs
```

如果您使用的是 Visual Studio IDE，那么您需要在项目属性中启用不安全代码。

步骤如下：

- 通过双击资源管理器（Solution Explorer）中的属性（properties）节点，打开项目属性（**project properties**）。
- 点击 **Build** 标签页。
- 选择选项"**Allow unsafe code**"。

C# 多线程

线程 被定义为程序的执行路径。每个线程都定义了一个独特的控制流。如果您的应用程序涉及到复杂的和耗时的操作，那么设置不同的线程执行路径往往是有益的，每个线程执行特定的工作。

线程是轻量级进程。一个使用线程的常见实例是现代操作系统中并行编程的实现。使用线程节省了 CPU 周期的浪费，同时提高了应用程序的效率。

到目前为止我们编写的程序是一个单线程作为应用程序的运行实例的单一的过程运行的。但是，这样子应用程序同时只能执行一个任务。为了同时执行多个任务，它可以被划分为更小的线程。

线程生命周期

线程生命周期开始于 System.Threading.Thread 类的对象被创建时，结束于线程被终止或完成执行时。

下面列出了线程生命周期中的各种状态：

- 未启动状态：当线程实例被创建但 Start 方法未被调用时的状况。
- 就绪状态：当线程准备好运行并等待 CPU 周期时的状况。
- 不可运行状态：下面的几种情况下线程是不可运行的：
    - 已经调用 Sleep 方法
    - 已经调用 Wait 方法
    - 通过 I/O 操作阻塞
- 死亡状态：当线程已完成执行或已中止时的状况。

主线程

在 C# 中，**System.Threading.Thread** 类用于线程的工作。它允许创建并访问多线程应用程序中的单个线程。进程中第一个被执行的线程称为主线程。

当 C# 程序开始执行时，主线程自动创建。使用 **Thread** 类创建的线程被主线程的子线程调用。您可以使用 Thread 类的 **CurrentThread** 属性访问线程。

下面的程序演示了主线程的执行：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class MainThreadProgram
    {
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";
            Console.WriteLine("This is {0}", th.Name);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is MainThread
```

Thread 类常用的属性和方法

下表列出了 **Thread** 类的一些常用的 属性：

属性	描述
CurrentContext	获取线程正在其中执行的当前上下文。
CurrentCulture	获取或设置当前线程的区域性。
CurrentPrinciple	获取或设置线程的当前负责人（对基于角色的安全性而言）。
CurrentThread	获取当前正在运行的线程。
CurrentUICulture	获取或设置资源管理器使用的当前区域性以便在运行时查找区域性特定的资源。
ExecutionContext	获取一个 ExecutionContext 对象，该对象包含有关当前线程的各种上下文的信息。
IsAlive	获取一个值，该值指示当前线程的执行状态。
IsBackground	获取或设置一个值，该值指示某个线程是否为后台线程。
IsThreadPoolThread	获取一个值，该值指示线程是否属于托管线程池。
ManagedThreadId	获取当前托管线程的唯一标识符。
Name	获取或设置线程的名称。
Priority	获取或设置一个值，该值指示线程的调度优先级。
ThreadState	获取一个值，该值包含当前线程的状态。

下表列出了 **Thread** 类的一些常用的 方法：

方法名	描述
public void Abort()	在调用此方法的线程上引发 ThreadAbortException，以开始终止此线程的过程。调用此方法通常会终止线程。
public static LocalDataStoreSlot AllocateDataSlot()	在所有的线程上分配未命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
public static LocalDataStoreSlot AllocateNamedDataSlot(string name)	在所有线程上分配已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
public static void BeginCriticalRegion()	通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常的影响可能会危害应用程序域中的其他任务。
public static void BeginThreadAffinity()	通知主机托管代码将要执行依赖于当前物理操作系统线程的标识的指令。
public static void EndCriticalRegion()	通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常仅影响当前任务。
public static void EndThreadAffinity()	通知主机托管代码已执行完依赖于当前物理操作系统线程的标识的指令。
public static void FreeNamedDataSlot(string name)	为进程中的所有线程消除名称与槽之间的关联。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
public static Object GetData(LocalDataStoreSlot slot)	在当前线程的当前域中从当前线程上指定的槽中检索值。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
public static AppDomain GetDomain()	返回当前线程正在其中运行的当前域。
public static AppDomain GetDomainID()	返回唯一的应用程序域标识符。
public static LocalDataStoreSlot GetNamedDataSlot(string name)	查找已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。

public void Interrupt()	中断处于 WaitSleepJoin 线程状态的线程。
public void Join()	在继续执行标准的 COM 和 SendMessage 消息泵处理期间，阻塞调用线程，直到某个线程终止为止。此方法有不同的重载形式。
public static void MemoryBarrier()	按如下方式同步内存存取：执行当前线程的处理器在对指令重新排序时，不能采用先执行 MemoryBarrier 调用之后的内存存取，再执行 MemoryBarrier 调用之前的内存存取的方式。
public static void ResetAbort()	取消为当前线程请求的 Abort。
public static void SetData(LocalDataStoreSlot slot, Object data)	在当前正在运行的线程上为此线程的当前域在指定槽中设置数据。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
public void Start()	开始一个线程。
public static void Sleep(int millisecondsTimeout)	让线程暂停一段时间。
public static void SpinWait(int iterations)	导致线程等待由 iterations 参数定义的时间量。
public static byte VolatileRead(ref byte address) public static double VolatileRead(ref double address) public static int VolatileRead(ref int address) public static Object VolatileRead(ref Object address)	读取字段值。无论处理器的数目或处理器缓存的状态如何，该值都是由计算机的任何处理器写入的最新值。此方法有不同的重载形式。这里只给出了一些形式。
public static void VolatileWrite(ref byte address, byte value) public static void VolatileWrite(ref double address, double value) public static void VolatileWrite(ref int address, int value) public static void VolatileWrite(ref Object address, Object value)	立即向字段写入一个值，以使该值对计算机中的所有处理器都可见。此方法有不同的重载形式。这里只给出了一些形式。
public static bool Yield()	导致调用线程执行准备好在当前处理器上运行的另一个线程。由操作系统选择要执行的线程。

创建线程

线程是通过扩展 Thread 类创建的。扩展的 Thread 类调用 **Start()** 方法来开始子线程的执行。

下面的程序演示了这个概念：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
```

管理线程

Thread 类提供了各种管理线程的方法。

下面的实例演示了 **sleep()** 方法的使用，用于在一个特定的时间暂停线程。

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
            // 线程暂停 5000 毫秒
            int sleepfor = 5000;
            Console.WriteLine("Child Thread Paused for {0} seconds",
                              sleepfor / 1000);
            Thread.Sleep(sleepfor);
            Console.WriteLine("Child thread resumes");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

销毁线程

Abort() 方法用于销毁线程。

通过抛出 **threadabortexception** 在运行时中止线程。这个异常不能被捕获，如果有 *finally* 块，控制会被送至 *finally* 块。

下面的程序说明了这点：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            try
            {

                Console.WriteLine("Child thread starts");
                // 计数到 10
                for (int counter = 0; counter <= 10; counter++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine(counter);
                }
                Console.WriteLine("Child Thread Completed");

            }
            catch (ThreadAbortException e)
            {
                Console.WriteLine("Thread Abort Exception");
            }
            finally
            {
                Console.WriteLine("Couldn't catch the Thread Exception");
            }

        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            // 停止主线程一段时间
            Thread.Sleep(2000);
            // 现在中止子线程
            Console.WriteLine("In Main: Aborting the Child thread");
            childThread.Abort();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
0
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```

W3School ASP.net 教程

来源：ASP.net教程

整理：飞龙

ASP.NET

经典 ASP - Active Server Pages（动态服务器页面）

ASP，全称 Active Server Pages（动态服务器页面），也被称为经典 ASP，是在1998年作为微软的第一个服务器端脚本引擎推出的。

ASP 是一种使得网页中的脚本在因特网服务器上被执行的技术。

ASP 页面的文件扩展名是 .asp，通常是用 VBScript 编写的。

如果您想学习经典 ASP，请访问我们的 经典 ASP 教程。

ASP.NET

ASP.NET 是新一代 ASP 。它与经典 ASP 是不兼容的，但 ASP.NET 可能包括经典 ASP。

ASP.NET 页面是经过编译的，这使得它们的运行速度比经典 ASP 快。

ASP.NET 具有更好的语言支持，有一大套的用户控件和基于 XML 的组件，并集成了用户身份验证。

ASP.NET 页面的扩展名是 .aspx，通常是用 VB (Visual Basic) 或者 C# (C sharp) 编写。

在 ASP.NET 中的控件可以用不同的语言（包括 C++ 和 Java）编写。

当浏览器请求 ASP.NET 文件时，ASP.NET 引擎读取文件，编译和执行脚本文件，并将结果以普通的 HTML 页面返回给浏览器。

ASP.NET Razor

Razor 是一种将服务器代码嵌入到 ASP.NET 网页中的新的、简单的标记语法，很像经典 ASP 。

Razor 具有传统的 ASP.NET 的功能，但更容易使用并且更容易学习。

ASP.NET 编程语言

本教程介绍了以下编程语言：

- Visual Basic (VB.NET)
- C# (发音：C sharp)

ASP.NET 服务器技术

本教程介绍了以下服务器技术

- Web Pages（Razor 语法)
- MVC（模型-视图-控制器）
- Web Forms（传统的 ASP.NET）

ASP.NET 开发工具

ASP.NET 支持以下开发工具：

- WebMatrix
- Visual Web Developer
- Visual Studio

在本教程中，Web Pages 教程使用了 WebMatrix， MVC 教程和 Web Forms 教程使用了 Visual Web Developer。

ASP.NET 文件扩展名

- 经典 ASP 文件的文件扩展名为 .asp
- ASP.NET 文件的文件扩展名为 .aspx
- Razor C# 语法的 ASP.NET 文件的文件扩展名为 .cshtml
- Razor VB 语法的 ASP.NET 文件的文件扩展名为 .vbhtml

Web Pages 教程

ASP.NET Web Pages - 教程

ASP.NET 是一个使用 HTML、CSS、JavaScript 和服务器脚本创建网页和网站的开发框架。

ASP.NET 支持三种不同的开发模式：
Web Pages（Web 页面）、MVC（Model View Controller 模型-视图-控制器）、Web Forms（Web 窗体）：

本教程介绍 **Web Pages**。

Web Pages
MVC
Web Forms

从何入手？

多数开发人员学习一个新技术，是从查看运行实例开始的。

通过"运行实例"轻松学习

我们的"运行实例"工具让 Web Pages 变得更简单易学。

它在运行实例的同时显示 ASP.NET 代码和 HTML 输出。

点击"运行实例"按钮来看看它是如何工作的：

Web Pages 实例

```
<html>
<body>
<h1>Hello Web Pages</h1>
<p>The time is @DateTime.Now</p>
</body>
</html>
```

运行实例？

什么是 Web Pages？

Web Pages 是三种创建 ASP.NET 网站和 Web 应用程序的编程模式中的一种。

其他两种编程模式是 Web Forms 和 MVC（Model View Controller 模型-视图-控制器）。

Web Pages 是开发 ASP.NET 网页最简单的开发模式。它提供了一种简单的方式来将 HTML、CSS、JavaScript 和服务器脚本结合起来：

- 容易学习，容易理解，容易使用
- 围绕着单一的网页创建
- 与 PHP 和经典 ASP 相似
- Visual Basic 或者 C# 的服务器脚本
- 全 HTML、CSS 和 JavaScript 控制

Web Pages 内置了数据库、视频、图形、社交媒体和其他更多的 Web Helpers，因此很容易扩展。

Web Pages 教程

如果您刚接触 ASP.NET，建议从 Web Pages 开始学习。

在我们的 Web Pages 教程中，您将学习到如何使用 VB(Visual Basic) 或者 C#(C sharp) 最新的 Razor 服务器标记语法将 HTML、CSS、JavaScript 和服务器代码结合起来。

您也可以学习如何使用具有可编程的 Web Helpers（包括数据库、视频、图形、社交媒体等等）来扩展您的网页。

Web Pages 实例

通过实例学习！

由于 ASP.NET 代码是在服务器上执行的，您不能在您的浏览器中查看代码。您只能看到普通的 HTML 页面输出。

在 w3cschool.cc 中，每个实例都会把隐藏的 ASP.NET 代码显示出来，这将让您更容易地理解它是如何工作的。

Web Pages 实例

Web Pages 参考手册

在本教程的最后，您将看到一套完整的 ASP.NET 参考手册，介绍了对象、组件、属性和方法。

Web Pages 参考手册

使用 WebMatrix

在本教程中，我们使用了 WebMatrix 。

WebMatrix 是一个简单但功能强大的，由微软专门为 Web Pages 量身定做的，免费的 ASP.NET 开发工具。

WebMatrix 包含：

- Web Pages 实例和模板
- 一种 Web 服务器语言（VB 或者 C# 的 Razor 服务器标记语法）
- 一种 Web 服务器（IIS Express）
- 一种数据库服务器（SQL Server Compact）
- 一个完整的 Web 开发框架（ASP.NET）

通过使用 WebMatrix ，您可以从一个空的网站和一个空白页面开始开发，或者您也可以使用"Web 应用程序库"中的开源应用程序进行二次开发。PHP 和 ASP.NET 应用程序很多都是开源的，比如 Umbraco、DotNetNuke、Drupal、Joomla、WordPress 等等。WebMatrix 也有内置安全性、搜索引擎优化和网络出版工具。

使用 WebMatrix 开发的技术和代码可以无缝地转化为完全专业化的 ASP.NET 应用程序。

如果您想尝试使用 WebMatrix ，请点击下面的链接进行安装：

http://www.microsoft.com/web/gallery/install.aspx?appid=WebMatrix

ASP.NET Web Pages - 添加 Razor 代码

在本教程中，我们将使用 C# 和 Visual Basic 代码的 Razor 标记。

什么是 Razor ？

- Razor 是一种将基于服务器的代码添加到网页中的标记语法
- Razor 具有传统 ASP.NET 标记的功能，但更容易使用并且更容易学习
- Razor 是一种服务器端标记语法，与 ASP 和 PHP 很像
- Razor 支持 C# 和 Visual Basic 编程语言

添加 Razor 代码

请记住上一章实例中的网页：

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title>Web Pages Demo</title>
</head>
<body>
<h1>Hello Web Pages</h1>
</body>
</html>
```

现在向实例中添加一些 Razor 代码：

实例

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title>Web Pages Demo</title>
</head>
<body>
<h1>Hello Web Pages</h1>
<p>The time is @DateTime.Now</p>
</body>
</html>
```

运行实例？

该页面中包含普通的 HTML 标记，除此之外，还添加了一个 @ 标识的 Razor 代码。

Razor 代码能够在服务器上实时地完成多有的动作，并将结果显示出来。（您可以指定格式化选项，否则只会显示默认项。）

主要的 Razor C# 语法规则

- Razor 代码块包含在 @{ ... } 中
- 内联表达式（变量和函数）以 @ 开头
- 代码语句用分号结束
- 变量使用 var 关键字声明
- 字符串用引号括起来
- C# 代码区分大小写
- C# 文件的扩展名是 .cshtml

C# 实例

```
<!-- Single statement block -->
@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
var greeting = "Welcome to our site!";
var weekDay = DateTime.Now.DayOfWeek;
var greetingMessage = greeting + " Today is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

运行实例？

主要的 Razor VB 语法规则

- Razor 代码块包含在 @Code ... End Code 中
- 内联表达式（变量和函数）以 @ 开头
- 变量使用 Dim 关键字声明
- 字符串用引号括起来
- VB 代码不区分大小写
- VB 文件的扩展名是 .vbhtml

实例

```
<!-- Single statement block -->
@Code dim myMessage = "Hello World" End Code

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@Code
dim greeting = "Welcome to our site!"
dim weekDay = DateTime.Now.DayOfWeek
dim greetingMessage = greeting & " Today is: " & weekDay
End Code

<p>The greeting is: @greetingMessage</p>
```

运行实例？

更多关于 C# 和 Visual Basic

如果您想学习更多关于 Razor、C#、Visual Basic 编程语言，请查看本教程的 Razor 部分。

ASP.NET Web Pages - 页面布局

通过 Web Pages，创建一个布局一致的网站是很容易的事。

一致的外观

在因特网上，您会发现很过网站都具有一致的外观和风格：

- 每个页面有相同的头部
- 每个页面有相同的底部
- 每个页面有相同的样式和布局

通过 Web Pages，您能非常高效地做到这点。您可以把重复使用的内容块（比如页面头部和底部）写在一个单独的文件中。

您还可以使用布局模板（布局文件）为站点的所有网页定义一致的布局。

Content Blocks（内容块）

许多网站都有一些内容是被显示在站点的每个页面中（比如页面头部和底部）。

通过 Web Pages，您可以使用 **@RenderPage()** 方法从不同的文件导入内容。

内容块（来自另一个文件）能被导入网页中的任何地方。内容块可以包含文本，标记和代码，就像任何普通的网页一样。

将共同的头部和底部写成单独的文件，这样会帮您节省大量的工作。您不必在每个页面中书写相同的内容，当内容有变动时，您只要修改头部或者底部文件，就可以看到站点中的每个页面的相应内容都已更新。

以下显示了它在代码中是如何呈现的：

实例

```
<html>
<body>
@RenderPage("header.cshtml")
<h1>Hello Web Pages</h1>
<p>This is a paragraph</p>
@RenderPage("footer.cshtml")
</body>
</html>
```

运行实例？

Layout Page（布局页）

在上一部分，您看到了，想在多个网页中显示相同内容是非常容易的。

另一种创建一致外观的方法是使用布局页。一个布局页包含了网页的结构，而不是内容。当一个网页（内容页）链接到布局页，它会根据布局页（模板）的结构进行显示。

布局页中使用 **@RenderBody()** 方法嵌入内容页，除此之外，它与一个正常的网页没有什么差别。

每个内容页都必须以布局指令开始。

以下显示了它在代码中是如何呈现的：

布局页：

```
<html>
<body>
<p>This is header text</p>
@RenderBody()
<p>&copy; 2012 W3CSchool. All rights reserved.</p>
</body>
</html>
```

任何网页：

```
@{Layout="Layout.cshtml";}

<h1>Welcome to W3CSchool.cc</h1>

<p>
Lorem ipsum dolor sit amet, consectetur adipisicing elit,sed do eiusmod tempor incididunt
</p>
```

运行实例？

D.R.Y. - Don't Repeat Yourself（不要自我重复）

通过 Content Blocks（内容块）和 Layout Pages（布局页）这两个 ASP.NET 工具，您可以让您的 Web 应用程序显示一致的外观。

这两个工具能帮您节省大量的工作，您不必再每个页面上重复相同的信息。集中的标记、样式和代码让您的 Web 应用程序更易于管理，更易于维护。

防止文件被浏览

在 ASP.NET 中，文件的名称以下划线开头，可以防止这些文件在网上被浏览。

如果您不想让您的内容块或者布局页被您的用户看到，可以重命名这些文件：

_header.cshtm

_footer.cshtml

_Layout.cshtml

隐藏敏感信息

在 ASP.NET 中，隐藏敏感信息（数据库密码、电子邮件密码等等）最通用的方法是将这些信息保存在一个名为"_AppStart"的单独的文件中。

_AppStart.cshtml

```
@{
WebMail.SmtpServer = "mailserver.example.com";
WebMail.EnableSsl = true;
WebMail.UserName = "username@example.com";
WebMail.Password = "your-password";
WebMail.From = "your-name-here@example.com";
}
```

ASP.NET Web Pages - 文件夹

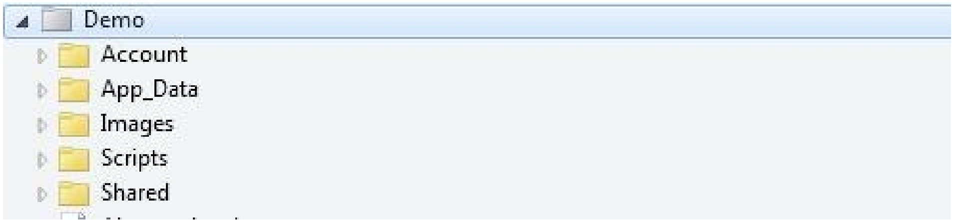
本章介绍有关文件夹和文件夹路径的知识。

在本章中，您将学到：

- 逻辑文件夹结构和物理文件夹结构
- 虚拟名称和物理名称
- Web URL 和 Web 路径

逻辑文件夹结构

下面是典型的 ASP.NET 网站文件夹结构：



- "Account" 文件夹包含登录和安全文件
- "App_Data" 文件夹包含数据库和数据文件
- "Images" 文件夹包含图片
- "Scripts" 文件夹包含浏览器脚本
- "Shared" 文件夹包含公共的文件（比如布局和样式文件）

物理文件夹结构

在上述网站中的"Images"文件夹在计算机上的物理文件夹结构可能如下：

C:\Documents\MyWebSites\Demo\Images

虚拟名称和物理名称

以上面的例子为例：

网站图片的虚拟名称可能是"Images/pic31.jpg"。

对应的物理名称是"C:\Documents\MyWebSites\Demo\Images\pic31.jpg"。

URL 和路径

URL 是用来访问网站中的文件： http://www.w3cschool.cc/html/html-tutorial.html

URL 对应于服务器上的物理文件：C:\MyWebSites\w3cschool\html\html-tutorial.html

虚拟路径是物理路径的一种简写表示。如果您使用虚拟路径，当您更改域名或者将您的网页移到其他服务器上时，您可以不用更新路径。

URL	http://www.w3cschool.cc/html/html-tutorial.html
服务器名称	w3cschool
虚拟路径	/html/html-tutorial.html
物理路径	C:\MyWebSites\w3cschool\html\html-tutorial.html

磁盘驱动器的根目录如下书写 C:，但是网站的根目录是 / （斜线）。

Web 文件夹的虚拟路径通常是与物理文件夹不相同。

在您的代码中，根据您的编码需要决定使用物理路径和和虚拟路径。

ASP.NET 文件夹路径有 3 种工具：~ 运算符、Server.MapPath 方法 和 Href 方法。

~ 运算符

使用 ~ 运算符，在编程代码中规定虚拟路径。

如果您使用 ~ 运算符，在您的站点迁移到其他不同的文件夹或者位置时，您可以不用更改您的任何代码：

```
var myImagesFolder = "~/images";
var myStyleSheet = "~/styles/StyleSheet.css";
```

Server.MapPath 方法

Server.MapPath 方法将虚拟路径（/index.html）转换成服务器能理解的物理路径（C:\Documents\MyWebSites\Demo\default.html）。

当您需要打开服务器上的数据文件时，您可以使用这个方法（只有提供完整的物理路径才能访问数据文件）：

```
var pathName = "~/dataFile.txt";
var fileName = Server.MapPath(pathName);
```

在本教程的下一章中，您会学到更多关于读取（和写入）服务器上的数据文件的知识。

Href 方法

Href 方法将代码中的使用的路径转换成浏览器可以理解的路径（浏览器无法理解 ~ 运算符）。

您可以使用 Href 方法创建资源（比如图像文件 和 CSS 文件）的路径。

一般会在 HTML 中的 <a>、<img> 和 <link> 元素中使用此方法:

```
@{var myStyleSheet = "~/Shared/Site.css";}

<!-- This creates a link to the CSS file. -->
<link rel="stylesheet" type="text/css" href="@Href(myStyleSheet)" />

<!-- Same as : -->
<link rel="stylesheet" type="text/css" href="/Shared/Site.css" />
```

Href 方法是 WebPage 对象的一种方法。

ASP.NET Web Pages - 全局页面

本章介绍全局页面 AppStart 和 PageStart。

在 Web 启动之前：_AppStart

大多数的服务器端代码是写在个人网页里边。例如，如果网页中包含输入表单，那么这个网页通常包含用来读取表单数据的服务器端代码。

然而，您可以通过在您的站点根目录下创建一个名为 _AppStart 的页面，这样在站点启动之前可以先启动代码执行。如果存在此页面，ASP.NET 会在站点中其它页面被请求时，优先运行这个页面。

_AppStart 的典型用途是启动代码和初始化全局数值（比如计数器和全局名称）。

注释 **1**：_AppStart 的文件扩展名与您的网页一致，比如：_AppStart.cshtml。

注释 **2**：_AppStart 有下划线前缀。因此，这些文件不可以直接浏览。

在每一个页面之前：_PageStart

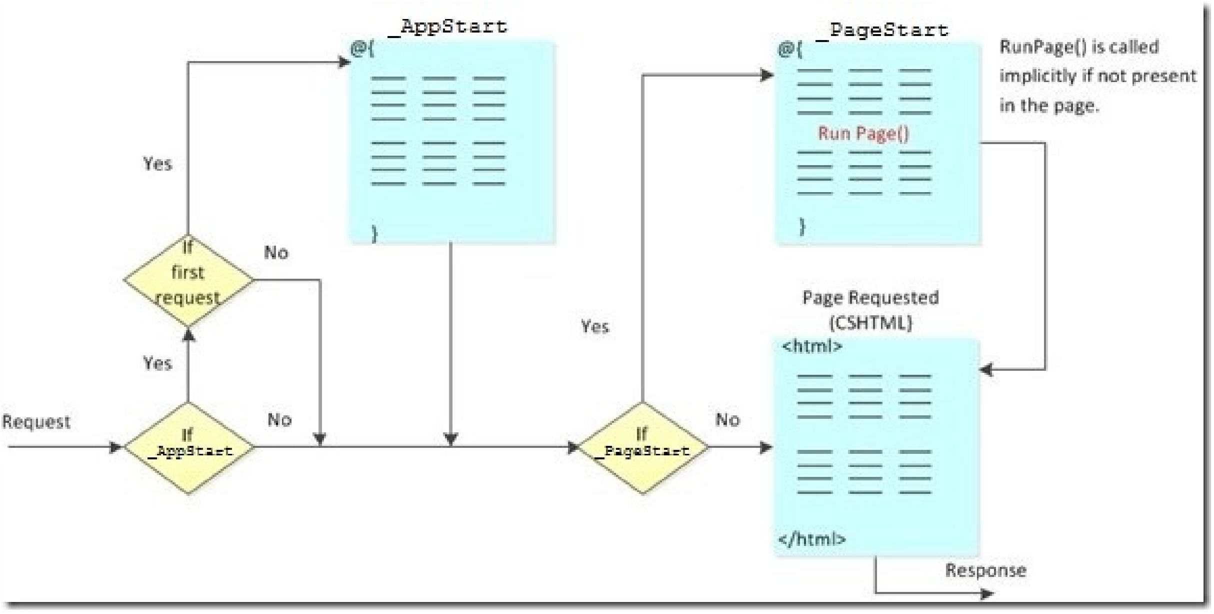
就像 _AppStart 在您的站点启动之前就运行一样，您可以编写在每个文件夹中的任何页面之前运行的代码。

对于您网站中的每个文件夹，您可以添加一个名为 _PageStart 的文件。

_PageStart 的典型用途是为一个文件夹中的所有页面设置布局页面，或者在运行某个页面之前检查用户是否已经登录。

它是如何工作的？

下图显示了它是如何工作的：



当接收到一个请求时，ASP.NET 会首先检查 _AppStart 是否存在。 如果 _AppStart 存在且这是站点接收到的第一个请求，则运行 _AppStart。

然后 ASP.NET 检查 _PageStart 是否存在。如果 _PageStart 存在，则在其它被请求的页面运行之前先运行 _PageStart。

您可以在 _PageStart 中调用 RunPage() 来指定被请求页面的运行位置。否则，默认情况下，被请求页面是在 _PageStart 运行之后才被运行。

ASP.NET Web Pages - HTML 表单

表单是 HTML 文档中放置输入控件（文本框、复选框、单选按钮、下拉列表）的部分。

创建一个 HTML 输入页面

Razor 实例

```
<html>
<body>
@{
if (IsPost) {
string companyname = Request["companyname"];
string contactname = Request["contactname"];
<p>You entered: <br />
Company Name: @companyname <br />
Contact Name: @contactname </p>
}
else
{
<form method="post" action="">
Company Name:<br />
<input type="text" name="CompanyName" value="" /><br />
Contact Name:<br />
<input type="text" name="ContactName" value="" /><br /><br />
<input type="submit" value="Submit" class="submit" />
</form>
}
}
</body>
</html>
```

运行实例？

Razor 实例 - 显示图像

假设在您的图像文件夹中有 3 张图像，您想根据用户的选择动态地显示图像。

这可以通过一段简单的 Razor 代码来实现。

如果在您的网站的图像文件夹中有一个名为 "Photo1.jpg" 的图像，您可以使用 HTML 的 <img> 元素来显示图像，如下所示：

下面的例子演示了如何显示用户从下列列表中选择的图像：

Razor 实例

```
@{
var imagePath="";
if (Request["Choice"] != null)
{imagePath="images/" + Request["Choice"];}
}
<!DOCTYPE html>
<html>
<body>
<h1>Display Images</h1>
<form method="post" action="">
I want to see:
<select name="Choice">
<option value="Photo1.jpg">Photo 1</option>
<option value="Photo2.jpg">Photo 2</option>
<option value="Photo3.jpg">Photo 3</option>
</select>
<input type="submit" value="Submit" />
@if (imagePath != "")
{
<p>
<img src="@imagePath" alt="Sample" />
</p>
}
</form>
</body>
</html>
```

运行实例？

实例解释

服务器创建了一个叫 **imagePath** 的变量。

HTML 页面有一个名为 **Choice** 的下拉列表（<select> 元素）。它允许用户根据自己的意愿选择一个名称（如 **Photo 1**），当页面被提交到 Web 服务器时，则传递了一个文件名（如 **Photo1.jpg**）。

Razor 代码通过 **Request["Choice"]** 读取 Choice 的值。如果通过代码构建的图像路径（images/Photo1.jpg）有效，就把图像路径赋值给变量 **imagePath**。

在 HTML 页面中，<img> 元素用来显示图像。当页面显示时，src 属性用来设置 imagePath 变量的值。

<img> 元素是在一个 if 块中，这是为了防止显示没有名称的图像，比如页面第一次被加载显示的时候。

ASP.NET Web Pages - 对象

Web Pages 经常是跟对象有关的。

Page 对象

您已经看到了一些在使用的 Page 对象方法：

```
@RenderPage("header.cshtml")

@RenderBody()
```

在前面的章节中，您已经看到了两个 Page 对象属性（isPost 和 Request）：

```
If (isPost) {

if (Request["Choice"] != null {
```

某些 Page 对象方法

方法	描述
href	使用指定的值创建 URL。
RenderBody()	呈现不在布局页命名区域的内容页的一部分。
RenderPage(*page*)	在另一个页面中呈现某一个页面的内容。
RenderSection(*section*)	呈现布局页命名区域的内容。
Write(*object*)	将对象作为 HTML 编码字符串写入。
WriteLiteral	写入对象时优先不使用 HTML 编码。

某些 Page 对象属性

属性	描述
isPost	如果客户端使用的 HTTP 数据传输方法是 POST 请求，则返回 true。
Layout	获取或者设置布局页面的路径。
Page	提供了对页面和布局页之间共享的数据的类似属性访问。
Request	为当前的 HTTP 请求获取 HttpRequest 对象。
Server	获取 HttpServerUtility 对象，该对象提供了网页处理方法。

Page 对象的 Page 属性

Page 对象的 Page 属性，提供了对页面和布局页之间共享的数据的类似属性访问。

您可以对 Page 属性使用（添加）您自己的属性：

- Page.Title
- Page.Version
- Page.anythingyoulike

页面属性是非常有用的。例如，在内容文件中设置页面标题，并在布局文件中使用：

Home.cshtml

```
@{
Layout="~/Shared/Layout.cshtml";
Page.Title="Home Page"
}

<h1>Welcome to W3CSchool.cc</h1>

<h2>Web Site Main Ingredients</h2>

<p>A Home Page (Default.cshtml)</p>
<p>A Layout File (Layout.cshtml)</p>
<p>A Style Sheet (Site.css)</p>
```

Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
<title>@Page.Title</title>
</head>
<body>
@RenderBody()
</body>
</html
```

ASP.NET Web Pages - 文件

本章介绍有关使用文本文件的知识。

使用文本文件

在前面的章节中，我们已经了解到网页数据是存储在数据库中的。

您也可以把站点数据存储在文本文件中。

用来存储数据的文本文件通常被称为平面文件。常见的文本文件格式是 .txt、.xml 和 .csv（逗号分隔值）。

在本章中，您将学习到：

- 如何从文本文件中读取并显示数据

手动添加一个文本文件

在下面的例子中，您将需要一个文本文件。

在您的网站上，如果没有 App_Data 文件夹，请创建一个。在 App_Data 文件夹中，创建一个名为 Persons.txt 的文件。

添加以下内容到文件中：

Persons.txt

```
George,Lucas
Steven,Spielberg
Alfred,Hitchcock
```

显示文本文件中的数据

下面的实例演示了如何显示一个文本文件中的数据：

实例

```
@{
var dataFile = Server.MapPath("~/App_Data/Persons.txt");
Array userData = File.ReadAllLines(dataFile);
}

<!DOCTYPE html>
<html>
<body>

<h1>Reading Data from a File</h1>
@foreach (string dataLine in userData)
{
foreach (string dataItem in dataLine.Split(','))
{@dataItem <text> </text>}
<br />
}
</body>
</html>
```

运行实例？

实例解释

使用 **Server.MapPath** 找到确切的文本文件的路径。

使用 **File.ReadAllLines** 打开文本文件，并读取文件中的所有行到一个数组中。

数组中的每个数据行中的数据项的数据被显示。

显示 Excel 文件中的数据

使用 Microsoft Excel，您可以将一个电子表格保存为一个逗号分隔的文本文件（.csv 文件）。此时，电子表格中的每一行保存为一个文本行，每个数据列由逗号分隔。

in可以使用上面的实例读取一个 Excel .csv 文件（只需将文件名改成相应的 Excel 文件的名称）。

ASP.NET Web Pages - 帮助器

Web 帮助器大大简化了 Web 开发和常见的编程任务。

ASP.NET 帮助器

ASP.NET 帮助器是通过几行简单的 Razor 代码即可访问的组件。

您可以使用存放在 .cshtml 文件中的 Razor 语法构建自己的帮助器，或者使用内建的 ASP.NET 帮助器。

在本教程接下来的章节中，您将学到如何使用 Razor 帮助器。

下面是一些有用的 Razor 帮助器的简短说明：

WebGrid 帮助器

WebGrid 帮助器简化了显示数据的方式：

- 自动创建一个 HTML 表格来显示数据
- 支持不同的格式化选项
- 支持数据分页显示（第一页、下一页、上一页、最后一页）
- 支持通过点击列表标题进行排序

Chart 帮助器

"Chart 帮助器" 能显示不同类型的带有多种格式化选项和标签的图表图像。

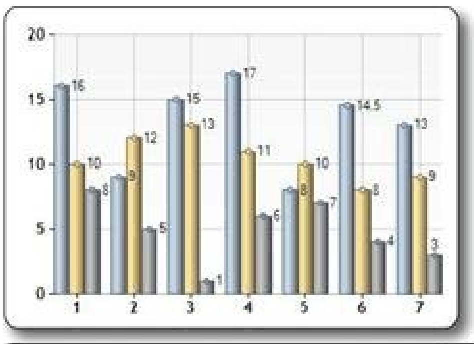
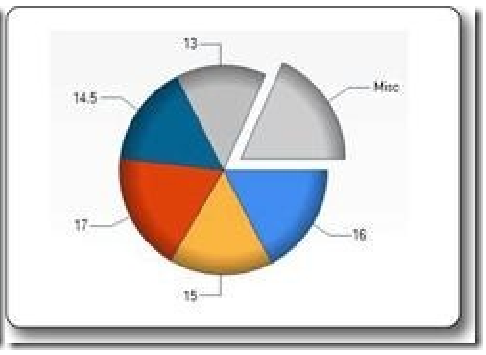



Chart 帮助器显示的数据来源可以是数组、数据库或者文件。

WebMail 帮助器

WebMail 帮助器提供了使用SMTP（Simple Mail Transfer Protocol 简单邮件传输协议）发送电子邮件的功能。

WebImage 帮助器

WebImage 帮助器提供了管理网页中图像的功能。

关键词：翻转、旋转、缩放、水印。

第三方帮助器

通过 Razor，您可以利用内建的或者第三方的帮助器来简化电子邮件、数据库、多媒体、社交网络以及很多其他问题（如导航和的网络安全）的使用。

安装帮助器

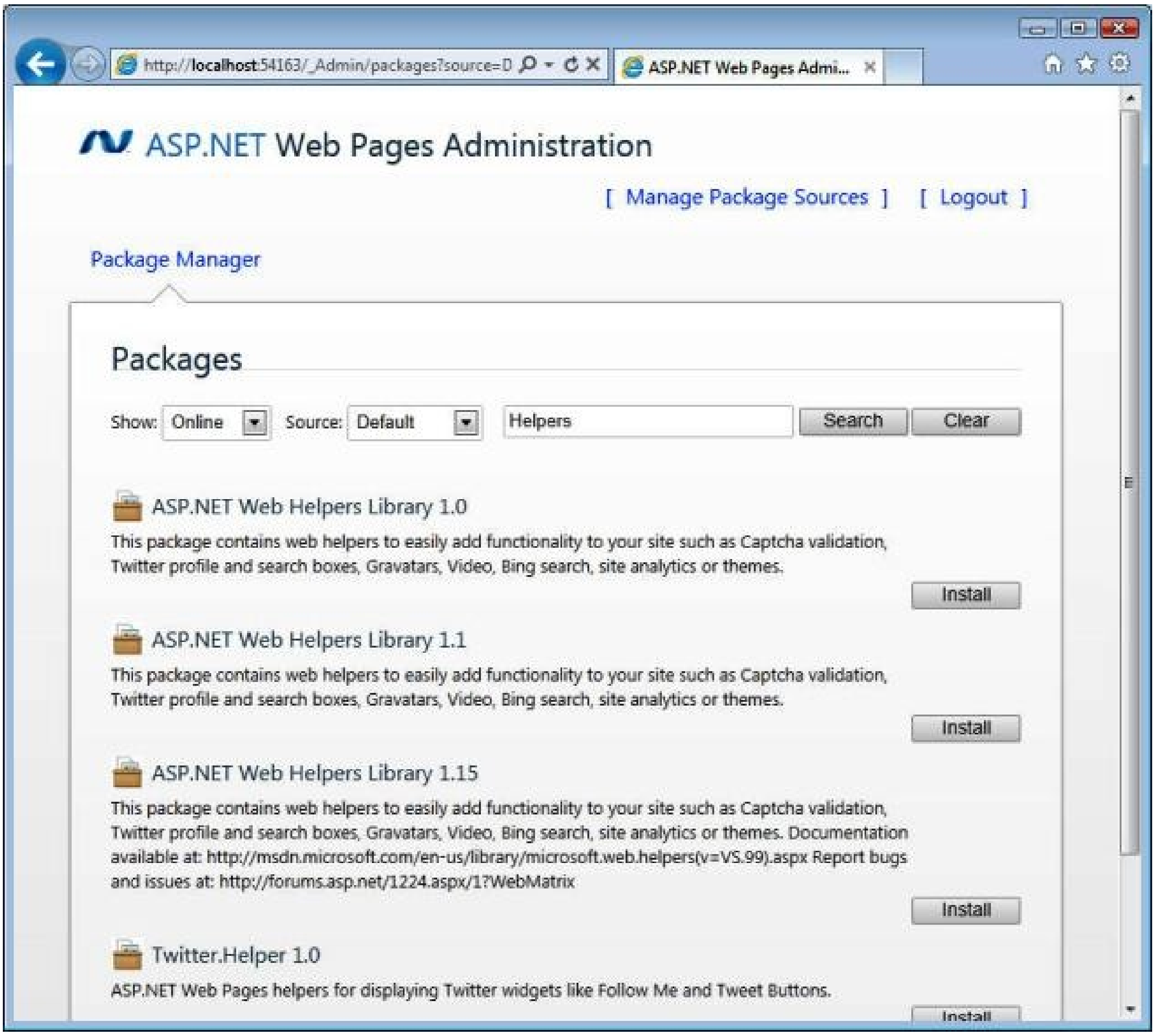
WebMatrix 已经包含了一些帮助器，您还可以手动安装其他的帮助器。

在 w3cschool.cc 的 WebPages 帮助器参考手册中，您可以看到一个便捷的参考手册，包含了内建帮助器和其他可以通过手动安装附加到 ASP.NET Web Helpers Library 工具包中的帮助器。

如果您在 WebMatrix 中创建了一个网站，请按照下面的步骤安装帮助器：

1. 在 WebMatrix 中，打开 **Site** 工作区。
2. 点击 **Web Pages Administration**。
3. 使用密码 * 登录到 Web Pages Administration。
4. 使用 搜索区 搜索帮助器。
5. 点击 **Install** 安装您所需的帮助器。

（* 如果您是第一次使用 Web Pages Administration，会提示您创建一个密码。）


http://localhost:54163/_Admin/packages?source=D
ASP.NET Web Pages Admi...
ASP.NET Web Pages Administration
[Manage Package Sources]
[Logout]
Package Manager
Packages
Show: Online
Source: Default
Helpers
Search
Clear
ASP.NET Web Helpers Library 1.0
This package contains web helpers to easily add functionality to your site such as Captcha validation, Twitter profile and search boxes, Gravatars, Video, Bing search, site analytics or themes.
Install
ASP.NET Web Helpers Library 1.1
This package contains web helpers to easily add functionality to your site such as Captcha validation, Twitter profile and search boxes, Gravatars, Video, Bing search, site analytics or themes.
Install
ASP.NET Web Helpers Library 1.15
This package contains web helpers to easily add functionality to your site such as Captcha validation, Twitter profile and search boxes, Gravatars, Video, Bing search, site analytics or themes. Documentation available at: http://msdn.microsoft.com/en-us/library/microsoft.web.helpers(v=VS.99).aspx Report bugs and issues at: http://forums.asp.net/1224.aspx/1?WebMatrix
Install
Twitter.Helper 1.0
ASP.NET Web Pages helpers for displaying Twitter widgets like Follow Me and Tweet Buttons.

ASP.NET Web Pages - WebGrid 帮助器

WebGrid - 众多有用的 ASP.NET Web 帮助器之一。

自己写的 HTML

在前面的章节中，您使用 Razor 代码显示数据库数据，所有的 HTML 标记都是手写的：

数据库实例

```
@{
var db = Database.Open("SmallBakery");
var selectQueryString = "SELECT * FROM Product ORDER BY Name";
}
<html>
<body>
<h1>Small Bakery Products</h1>
<table>
<tr>
<th>Id</th>
<th>Product</th>
<th>Description</th>
<th>Price</th>
</tr>
@foreach(var row in db.Query(selectQueryString))
{
<tr>
<td>@row.Id</td>
<td>@row.Name</td>
<td>@row.Description</td>
<td align="right">@row.Price</td>
</tr>
}
</table>
</body>
</html>
```

运行实例？

使用 WebGrid 帮助器

WebGrid 帮助器提供了一种更简单的显示数据的方法。

WebGrid 帮助器：

- 自动创建一个 HTML 表格来显示数据
- 支持不同的格式化选项
- 支持数据分页显示
- 支持通过点击列表标题进行排序

WebGrid 实例

```
@{
var db = Database.Open("SmallBakery") ;
var selectQueryString = "SELECT * FROM Product ORDER BY Id";
var data = db.Query(selectQueryString);
var grid = new WebGrid(data);
}
<html>
<head>
<title>Displaying Data Using the WebGrid Helper</title>
</head>
<body>
<h1>Small Bakery Products</h1>
<div id="grid">
@grid.GetHtml()
</div>
</body>
</html>
```

运行实例？

ASP.NET Web Pages - Chart 帮助器

Chart 帮助器 - 众多有用的 ASP.NET Web 帮助器之一。

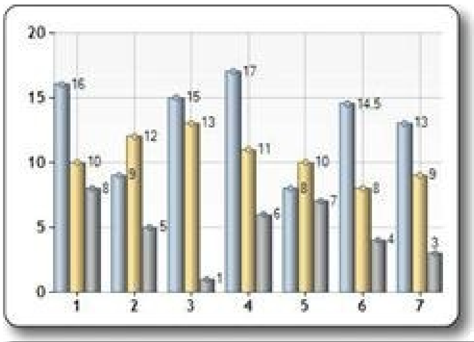
Chart 帮助器

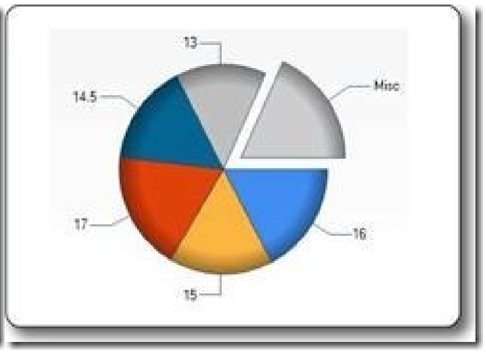
在前面的章节中，您已经学习了如何使用 ASP.NET 的 "帮助器"。

前面已经介绍了如何使用 "WebGrid 帮助器" 在网格中显示数据。

本章介绍如何使用 "Chart 帮助器" 以图形化的形式显示数据。

"Chart 帮助器" 可以创建不同类型的带有多种格式化选项和标签的图表图像。它可以创建面积图、条形图、柱形图、折线图、饼图等标准图表，也可以创建像股票图表这样的更专业的图表。





在图表中显示的数据可以是来自一个数组，一个数据库，或者一个文件中的数据。

根据数组创建图表

下面的实例显示了根据数组数据显示图表所需的代码：

实例

```
@{
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Employees")
.AddSeries(chartType: "column",
xValue: new[] { "Peter", "Andrew", "Julie", "Mary", "Dave" },
yValues: new[] { "2", "6", "4", "5", "3" })
.Write();
}
```

运行实例？

- **new Chart** 创建一个新的图表对象并且设置它的宽度和高度
- **AddTitle** 方法指定了图表的标题
- **AddSeries** 方法向图表中增加数据
- **chartType** 参数定义图表的类型
- **xValue** 参数定义 x 轴的名称
- **yValues** 参数定义 y 轴的名称
- **Write()** 方法显示图表

根据数据库创建图表

您可以执行一个数据库查询，然后使用查询结果中的数据来创建一个图表：

实例

```
@{
var db = Database.Open("SmallBakery");
var dbdata = db.Query("SELECT Name, Price FROM Product");
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Product Sales")
.DataBindTable(dataSource: dbdata, xField: "Name")
.Write();
}
```

运行实例？

- **var db = Database.Open** 打开数据库（将数据库对象赋值给变量 db）
- **var dbdata = db.Query** 执行数据库查询并保存结果在 dbdata 中
- **new Chart** 创建一个新的图表对象并且设置它的宽度和高度
- **AddTitle** 方法指定了图表的标题
- **DataBindTable** 方法将数据源绑定到图表
- **Write()** 方法显示图表

除了使用 DataBindTable 方法之外，另一种方法是使用 AddSeries（见前面的实例）。DataBindTable 更容易使用，但是 AddSeries 更加灵活，因为您可以更明确地指定图表和数据：

实例

```
@{
var db = Database.Open("SmallBakery");
var dbdata = db.Query("SELECT Name, Price FROM Product");
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Product Sales")
.AddSeries(chartType:"Pie",
xValue: dbdata, xField: "Name",
yValues: dbdata, yFields: "Price")
.Write();
}
```

运行实例？

根据 XML 数据创建图表

第三种创建图表的方法是使用 XML 文件作为图表的数据：

实例

```
@using System.Data;

@{
var dataSet = new DataSet();
dataSet.ReadXmlSchema(Server.MapPath("data.xsd"));
dataSet.ReadXml(Server.MapPath("data.xml"));
var dataView = new DataView(dataSet.Tables[0]);
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Sales Per Employee")
.AddSeries("Default", chartType: "Pie",
xValue: dataView, xField: "Name",
yValues: dataView, yFields: "Sales")
.Write();}
}
```

运行实例？

ASP.NET Web Pages - WebMail 帮助器

WebMail 帮助器 - 众多有用的 ASP.NET Web 帮助器之一。

WebMail 帮助器

WebMail 帮助器让发送邮件变得更简单，它按照 SMTP（Simple Mail Transfer Protocol 简单邮件传输协议）从 Web 应用程序发送邮件。

前提：电子邮件支持

为了演示如何使用电子邮件，我们将创建一个输入页面，让用户提交一个页面到另一个页面，并发送一封关于支持问题的邮件。

第一：编辑您的 AppStart 页面

如果在本教程中您已经创建了 Demo 应用程序，那么您已经有一个名为 _AppStart.cshtml 的页面，内容如下：

_AppStart.cshtml

```
@{
WebSecurity.InitializeDatabaseConnection("Users", "UserProfile", "UserId", "Email", true)
}
```

要启动 WebMail 帮助器，向您的 AppStart 页面中增加如下所示的 WebMail 属性：

_AppStart.cshtml

```
@{
WebSecurity.InitializeDatabaseConnection("Users", "UserProfile", "UserId", "Email", true)
WebMail.SmtpServer = "smtp.example.com";
WebMail.SmtpPort = 25;
WebMail.EnableSsl = false;
WebMail.UserName = "support@example.com";
WebMail.Password = "password-goes-here";
WebMail.From = "john@example.com";
}
```

属性解释：

SmtpServer: 用于发送电子邮件的 SMTP 服务器的名称。

SmtpPort: 服务器用来发送 SMTP 事务（电子邮件）的端口。

EnableSsl: 如果服务器使用 SSL（Secure Socket Layer 安全套接层）加密，则值为 true。

UserName: 用于发送电子邮件的 SMTP 电子邮件账户的名称。

Password: SMTP 电子邮件账户的密码。

From: 在发件地址栏显示的电子邮件（通常与 UserName 相同）。

第二：创建一个电子邮件输入页面

接着创建一个输入页面，并将它命名为 Email_Input：

Email_Input.cshtml

```
<!DOCTYPE html>
<html>
<body>
<h1>Request for Assistance</h1>

<form method="post" action="EmailSend.cshtml">
<label>Username:</label>
<input type="text name="customerEmail" />
<label>Details about the problem:</label>
<textarea name="customerRequest" cols="45" rows="4"></textarea>
<p><input type="submit" value="Submit" /></p>
</form>

</body>
</html>
```

输入页面的目的是手机信息，然后提交数据到可以将信息作为电子邮件发送的一个新的页面。

第三：创建一个电子邮件发送页面

接着创建一个用来发送电子邮件的页面，并将它命名为 Email_Send：

Email_Send.cshtml

```
@{ // Read input
var customerEmail = Request["customerEmail"];
var customerRequest = Request["customerRequest"];
try
{
// Send email
WebMail.Send(to:"someone@example.com", subject: "Help request from - " + customerEmail, b
}
catch (Exception ex )
{
<text>@ex</text>
}
}
```

想了解更多关于 ASP.NET Web Pages 应用程序发送电子邮件的信息，请查阅：WebMail 对象参考手册。

ASP.NET Web Pages - PHP

PHP 开发人员请注意，Web Pages 可以用 PHP 编写。

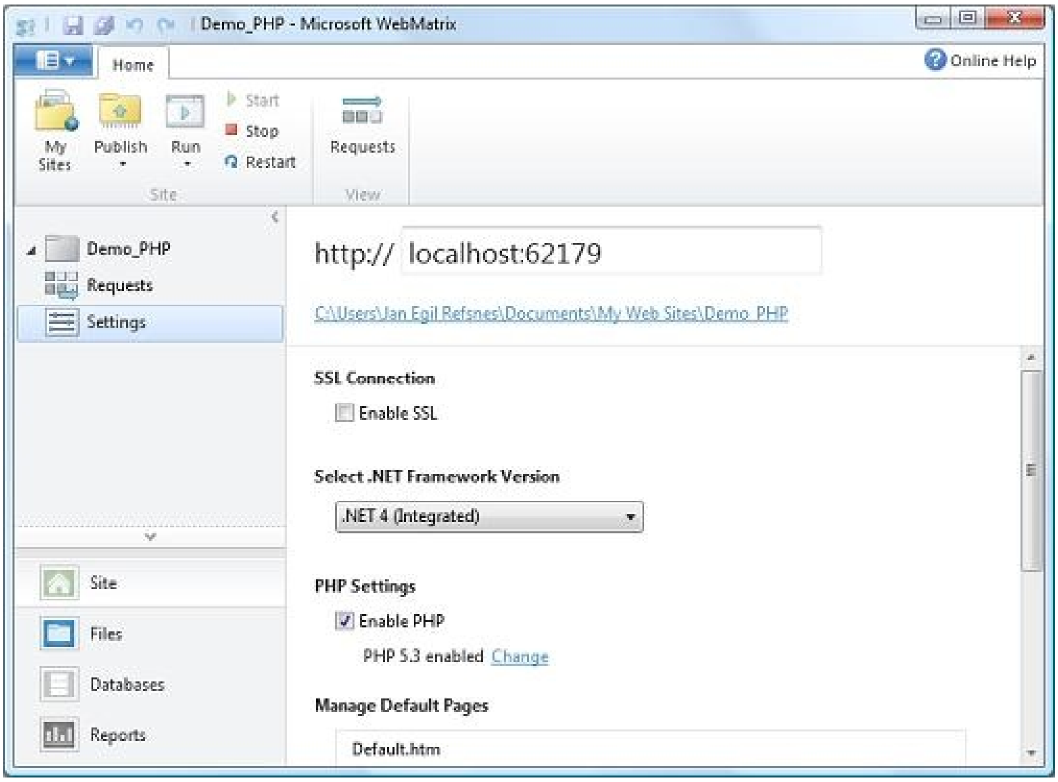
WebMatrix 支持 PHP

乍一看，认为 WebMatrix 只支持微软的技术。这是不正确的。在 WebMatrix 中，您能编写完整的 PHP 应用程序。

创建一个 PHP 站点

在ASP.NET Web Pages - 创建一个网站章节中，您已经创建了一个名为 "Demo" 的空网站，带有一个类型为 "CSHTML" 的空页面。

重复一遍创建的过程，创建一个名为 "Demo_PHP" 的空站点，勾选 "Enable PHP"（如下图所示），创建一个 PHP 类型的空白页，并将它命名 "index.php"，这样您就创建好了您的第一个 PHP 站点。



创建一个 PHP 页面

将下面的代码复制到 "index.php" 文件中：

index.php

```
<!DOCTYPE html>
<html>
<body>

<?php
phpinfo();
?>

</body>
</html>
```

运行文件，看看 PHP 页面的演示。

ASP.NET Web Pages - 发布网站

学习如何在不使用 WebMatrix 的情况下发布 Web Pages 应用程序。

在不使用 WebMatrix 的情况下发布您的应用程序

通过在 WebMatrix（或者 Visual Studio）中使用发布命令，可以发布一个 ASP.NET Web Pages 应用程序到远程服务器上。

此功能会复制所有您的应用程序文件、cshtml页面、图像以及用于 Web Pages、Razor、Helpers、SQL Server Compact（如果使用数据库）所有必需的 DLL 文件。

有时您不想使用 WebMatrix 发布您的应用程序。也许是因为您的托管服务提供商只支持 FTP，也许您已经有一个基于经典 ASP 的网站，也许您想自己复制所有的文件，也许您想使用 Front Page、Expression Web 等其他一些发布软件。

您会遇到问题吗？是的，会的。但是您有办法解决它。

要执行网站复制，您必须知道如何引用正确的文件，哪些 DLL 文件需要复制，并在何处存储它们。

请按照下列步骤操作：

1. 使用最新版本的 ASP.NET

在您继续操作之前，请确保您的主机运行的是最新版的 ASP.NET（4.0 或者 4.5）。

2. 复制 Web 文件夹

从您的开发计算机上复制您的网站（所有文件夹和内容）到远程主机（服务器）上的应用程序文件夹中。

> 如果您的应用程序中包含数据，不要复制数据（详见下面的第 4 点）。

3. 复制 DLL 文件

确保您的远程主机上的 bin 文件夹中包含了和您开发计算机上相同的 dll 文件。

复制 bin 文件夹之后，它应该包含以下文件：

Microsoft.Web.Infrastructure.dll
NuGet.Core.dll
System.Web.Helpers.dll
System.Web.Razor.dll
System.Web.WebPages.Administration.dll
System.Web.WebPages.Deployment.dll
System.Web.WebPages.dll
System.Web.WebPages.Razor.dll
WebMatrix.Data.dll
WebMatrix.WebData

4. 复制您的数据

如果您的应用程序包含数据或者数据库。例如 SQL Server Compact 数据库（在 App_Data 文件夹中的一个 .sdf 文件），请考虑以下几点：

您是否希望发布您的测试数据到远程服务器上？

大多数时候一般是不希望。

如果在您的开发计算机上有测试数据，它将覆盖您的远程主机上的生产数据。

如果您一定要复制 SQL 数据库（.sdf 文件），那么您应该删除数据库中的所有数据，然后从您的开发计算机上复制一个空的 .sdf 文件到服务器上。

就是这样。**GOOD LUCK**！

Razor 教程

ASP.NET Razor - 标记

Razor 不是一种编程语言。它是服务器端的标记语言。

什么是 Razor？

Razor 是一种标记语法，可以让您将基于服务器的代码（Visual Basic 和 C#）嵌入到网页中。

基于服务器的代码可以在网页传送给浏览器时，创建动态 Web 内容。当一个网页被请求时，服务器在返回页面给浏览器之前先执行页面中的基于服务器的代码。通过服务器的运行，代码能执行复杂的任务，比如进入数据库。

Razor 是基于 ASP.NET 的，是为创建 Web 应用程序而设计的。它具有传统 ASP.NET 的功能，但更容易使用并且更容易学习。

Razor 语法

Razor 使用了与 PHP 和经典 ASP 相似的语法。

Razor：

```
<ul>
@for (int i = 0; i < 10; i++) {
<li>@i</li>
}
</ul>
```

PHP：

```
<ul>
<?php
for ($i = 0; $i < 10; $i++) {
echo("<li>$i</li>");
}
?>
</ul>
```

Web Forms（经典 ASP）：

```
<ul>
<% for (int i = 0; i < 10; i++) { %>
<li><% =i %></li>
<% } %>
</ul>
```

Razor 帮助器

ASP.NET 帮助器是通过几行简单的 Razor 代码即可访问的组件。

您可以使用 Razor 语法构建自己的帮助器，或者使用内建的 ASP.NET 帮助器。

下面是一些有用的 Razor 帮助器的简短说明：

- Web Grid（Web 网格）
- Web Graphics（Web 图形）
- Google Analytics（Google 分析）
- Facebook Integration（Facebook 集成）
- Twitter Integration（Twitter 集成）
- Sending Email（发送电子邮件）
- Validation（验证）

Razor 编程语言

Razor 支持 C# (C sharp) 和 VB (Visual Basic)。

ASP.NET Razor - C# 和 VB 代码语法

Razor 同时支持 C# (C sharp) 和 VB (Visual Basic)。

主要的 Razor C# 语法规则

- Razor 代码块包含在 @{ ... } 中
- 内联表达式（变量和函数）以 @ 开头
- 代码语句用分号结束
- 变量使用 var 关键字声明
- 字符串用引号括起来
- C# 代码区分大小写
- C# 文件的扩展名是 .cshtml

C# 实例

```
<!-- Single statement block -->
@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
var greeting = "Welcome to our site!";
var weekDay = DateTime.Now.DayOfWeek;
var greetingMessage = greeting + " Here in Huston it is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

运行实例？

主要的 Razor VB 语法规则

- Razor 代码块包含在 @Code ... End Code 中
- 内联表达式（变量和函数）以 @ 开头
- 变量使用 Dim 关键字声明
- 字符串用引号括起来
- VB 代码不区分大小写
- VB 文件的扩展名是 .vbhtml

实例

```
<!-- Single statement block -->
@Code dim myMessage = "Hello World" End Code

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@Code
dim greeting = "Welcome to our site!"
dim weekDay = DateTime.Now.DayOfWeek
dim greetingMessage = greeting & " Here in Huston it is: " & weekDay
End Code

<p>The greeting is: @greetingMessage</p>
```

运行实例？

它是如何工作的？

Razor 是一种将服务器代码嵌入在网页中的简单的编程语法。

Razor 语法是基于 ASP.NET 框架，专门用于创建 Web 应用程序的部分 Microsoft.NET 框架。

Razor 语法支持所有 ASP.NET 的功能，但是使用的是一种简化语法，对初学者而言更容易学习，对专家而言更有效率的。

Razor 网页可以被描述成带一下两种类型内容的 HTML 网页： HTML 内容和 Razor 代码。

当服务器读取页面时，它首先运行 Razor 代码，然后再发送 HTML 页面到浏览器。在服务器上执行的代码能够执行一些在浏览器上不能完成的任务，比如，访问服务器数据库。服务器代码能创建动态的 HTML 内容，然后发送到浏览器。从浏览器上看，服务器代码生成的 HTML 与静态的 HTML 内容没有什么不同。

带 Razor 语法的 ASP.NET 网页有特殊的文件扩展名 cshtml（Razor C#）或者 vbhtml（Razor VB）。

使用对象

服务器编码往往涉及到对象。

"Date" 对象是一个典型的内置的 ASP.NET 对象，但对象也可以是自定义的，一个网页，一个文本框，一个文件，一个数据库记录，等等。

对象有用于执行的方法。一个数据库记录可能有一个 "Save" 方法，一个图像对象可能有一个 "Rotate" 方法，一个电子邮件对象可能有一个 "Send" 方法，等等。

对象也有用于描述各自特点的属性。一个数据库记录可能有 FirstName 和 LastName 属性。

ASP.NET Date 对象有一个 Now 属性（写成 Date.Now），Now 属性有一个 Day 属性（写成 Date.Now.Day）。下面实例演示了如何访问 Data 对象的一些属性：

实例

```
<table border="1">
<tr>
<th width="100px">Name</th>
<td width="100px">Value</td>
</tr>
<tr>
<td>Day</td><td>@DateTime.Now.Day</td>
</tr>
<tr>
<td>Hour</td><td>@DateTime.Now.Hour</td>
</tr>
<tr>
<td>Minute</td><td>@DateTime.Now.Minute</td>
</tr>
<tr>
<td>Second</td><td>@DateTime.Now.Second</td>
</tr>
</td>
</table>
```

运行实例？

If 和 Else条件

动态网页的一个重要特点是，您可以根据条件决定做什么。

做到这一点的常用方法是使用 if ... else 语句：

实例

```
@{
var txt = "";
if(DateTime.Now.Hour > 12)
{txt = "Good Evening";}
else
{txt = "Good Morning";}
}
<html>
<body>
<p>The message is @txt</p>
</body>
</html>
```

运行实例？

读取用户输入

动态网页的另一个重要特点是，您可以读取用户输入。

输入是通过 Request[] 功能读取的，并且传送输入数据是经过 IsPost 条件判断的：

实例

```
@{
var totalMessage = "";
if(IsPost)
{
var num1 = Request["text1"];
var num2 = Request["text2"];
var total = num1.AsInt() + num2.AsInt();
totalMessage = "Total = " + total;
}
}
<html>
<body style="background-color: beige; font-family: Verdana, Arial;">
<form action="" method="post">
<p><label for="text1">First Number:</label><br>
<input type="text" name="text1" /></p>
<p><label for="text2">Second Number:</label><br>
<input type="text" name="text2" /></p>
<p><input type="submit" value=" Add " /></p>
</form>
<p>@totalMessage</p>
</body>
</html>
```

运行实例？

ASP.NET Razor - C# 变量

变量是用来存储数据的命名实体。

变量

变量是用来存储数据的。

一个变量的名称必须以字母字符开头，并且不能包含空格或者保留字符。

一个变量可以是一个指定的类型，表示它所存储的数据类型。string 变量存储字符串值（"Welcome to W3CSchool.cc"），integer 变量存储数字值（103），date 变量存储日期值，等等。

变量使用 var 关键字声明，或通过使用类型（如果您想声明类型）声明，但是 ASP.NET 通常能自动确定数据类型。

实例

```
// Using the var keyword:
var greeting = "Welcome to W3CSchool.cc";
var counter = 103;
var today = DateTime.Today;

// Using data types:
string greeting = "Welcome to W3CSchool.cc";
int counter = 103;
DateTime today = DateTime.Today;
```

数据类型

下面列出了常用的数据类型：

类型	描述	实例
int	整数（全数字）	103, 12, 5168
float	浮点数	3.14, 3.4e38
decimal	十进制数字（高精度）	1037.196543
bool	布尔值	true, false
string	字符串	"Hello W3CSchool.cc", "John"

运算符

运算符告诉 ASP.NET 在表达式中执行什么样的命令。

C# 语言支持多种运算符。下面列出了常用的运算符：

<table>
<tr><th>运算符</th><th>描述</th><th>实例</th></tr>
<tr><td>=</td><td>给一个变量赋值。</td><td>i=6</td></tr>
<tr><td>+
-
*
/</td><td>加上一个值或者一个变量。 减去一个值或者一个变量。 乘以一个值或者一个变量。 除以一个值或者一个变量。</td><td>i=5+5 i=5-5 i=5*5 i=5/5</td></tr>
<tr><td>+=
-=</td><td>变量递增。 变量递减。</td><td>i += 1 i -= 1</td></tr>
<tr><td>==</td><td>相等。如果值相等则返回 true。</td><td>if (i==10)</td></tr>
<tr><td>!=</td><td>不等。如果值不等则返回 true。</td><td>if (i!=10)</td></tr>
<tr><td><
>
<=
>=</td><td>小于。 大于。 小于等于。 大于等于。</td><td>if (i<10) if (i>10) if (i<=10) if (i>=10)</td></tr>
<tr><td>+</td><td>连接字符串（一系列互相关联的事物）。</td><td>"w3" + "schools"</td></tr>
<tr><td>.</td><td>点号。分隔对象和方法。</td><td>DateTime.Hour</td></tr>
<tr><td>()</td><td>圆括号。将值进行分组。</td><td>(i+5)</td></tr>
<tr><td>()</td><td>圆括号。传递参数。</td><td>x=Add(i,5)</td></tr>
<tr><td>[]</td><td>中括号。访问数组或者集合的值。</td><td>name[3]</td></tr>
<tr><td>!</td><td>非。真/假取反。</td><td>if (!ready)</td></tr>
<tr><td>&&
||</td><td>逻辑与。 逻辑或。</td><td>if (ready && clear)
if (ready || clear)</td></tr>
</table>

转换数据类型

从一种数据类型转换到另一种数据类型，有时候是很有用的。

最常见的例子是将字符串输入转换为另一种类型，如整数或者日期。

一般规则下，都是将用户输入看做字符串处理，即使用户输入了数字。因此数值输入必须被转换成数字，然后才能将其用于计算。

下面列出了常用的转换方法：

方法	描述	实例
AsInt() IsInt()	转换字符串为整数。	if (myString.IsInt()) {myInt=myString.AsInt();}
AsFloat() IsFloat()	转换字符串为浮点数。	if (myString.IsFloat()) {myFloat=myString.AsFloat();}
AsDecimal() IsDecimal()	转换字符串为十进制数。	if (myString.IsDecimal()) {myDec=myString.AsDecimal();}
AsDateTime() IsDateTime()	转换字符串为 ASP.NET DateTime 类型。	myString="10/10/2012"; myDate=myString.AsDateTime();
AsBool() IsBool()	转换字符串为布尔值。	myString="True"; myBool=myString.AsBool();
ToString()	转换任何数据类型为字符串。	myInt=1234; myString=myInt.ToString();

ASP.NET Razor - C# 循环和数组

语句在循环中会被重复执行。

For 循环

如果您需要重复执行相同的语句，您可以设定一个循环。

如果您知道要循环的次数，您可以使用 **for** 循环。这种类型的循环在向上计数或向下计数时特别有用：

实例

```
<html>
<body>
@for(var i = 10; i < 21; i++)
{<p>Line @i</p>}
</body>
</html>
```

运行实例？

For Each 循环

如果您使用的是集合或者数组，您会经常用到 **for each** 循环。

集合是一组相似的对象，for each 循环可以遍历集合直到完成。

下面的实例中，遍历 ASP.NET Request.ServerVariables 集合。

实例

```
<html>
<body>
<ul>
@foreach (var x in Request.ServerVariables)
{<li>@x</li>}
</ul>
</body>
</html>
```

运行实例？

While 循环

while 循环是一个通用的循环。

while 循环以 while 关键字开始，后面紧跟着括号，您可以在括号里规定循环将持续多久，然后是重复执行的代码块。

while 循环通常会设定一个递增或者递减的变量用来计数。

下面的实例中，+= 运算符在每执行一次循环时给变量 i 的值加 1。

实例

```
<html>
<body>
@{
var i = 0;
while (i < 5)
{
i += 1;
<p>Line #@i</p>
}
}
</body>
</html>
```

运行实例？

数组

当您要存储多个相似变量但又不想为每个变量都创建一个独立的变量时，可以使用数组来存储：

实例

```
@{
string[] members = {"Jani", "Hege", "Kai", "Jim"};
int i = Array.IndexOf(members, "Kai")+1;
int len = members.Length;
string x = members[2-1];
}
<html>
<body>
<h3>Members</h3>
@foreach (var person in members)
{
<p>@person</p>
}
<p>The number of names in Members are @len</p>
<p>The person at position 2 is @x</p>
<p>Kai is now in position @i</p>
</body>
</html>
```

运行实例？

ASP.NET Razor - C# 逻辑条件

编程逻辑：根据条件执行代码。

If 条件

C# 允许根据条件执行代码。

使用 **if** 语句来判断条件。根据判断结果，if 语句返回 true 或者 false：

- if 语句开始一个代码块
- 条件写在括号里
- 如果条件为真，大括号内的代码被执行

实例

```
@{var price=50;}
<html>
<body>
@if (price>30)
{
<p>The price is too high.</p>
}
</body>
</html>
```

运行实例？

Else 条件

if 语句可以包含 **else** 条件。

else 条件定义了当条件为假时被执行的代码。

实例

```
@{var price=20;}
<html>
<body>
@if (price>30)
{
<p>The price is too high.</p>
}
else
{
<p>The price is OK.</p>
}
</body>
</html>
```

运行实例？

注释：在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。else 条件覆盖了除 if 条件之外的"其他所有情况"。

Else If 条件

多个条件判断可以使用 **else if** 条件：

实例

```
@{var price=25;}
<html>
<body>
@if (price>=30)
{
<p>The price is high.</p>
}
else if (price>20 && price<30)
{
<p>The price is OK.</p>
}
else
{
<p>The price is low.</p>
}
</body>
</html>
```

运行实例？

在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。

如果第一个条件不为真且第二个条件为真，else if 块的代码将会被执行。

else if 条件的数量不受限制。

如果 if 和 else if 条件都不为真，最后的 else 块（不带条件）覆盖了"其他所有情况"。

Switch 条件

switch 块可以用来测试一些单独的条件：

实例

```
@{
var weekday=DateTime.Now.DayOfWeek;
var day=weekday.ToString();
var message="";
}
<html>
<body>
@switch(day)
{
case "Monday":
message="This is the first weekday.";
break;
case "Thursday":
message="Only one day before weekend.";
break;
case "Friday":
message="Tomorrow is weekend!";
break;
default:
message="Today is " + day;
break;
}
<p>@message</p>
</body>
</html>
```

运行实例？

测试值（day）是写在括号中。每个单独的测试条件都有一个以分号结束的 case 值和以 break 语句结束的任意数量的代码行。如果测试值与 case 值相匹配，相应的代码行被执行。

switch 块有一个默认的情况（default:），当所有的指定的情况都不匹配时，它覆盖了"其他所有情况"。

ASP.NET Razor - VB 变量

变量是用来存储数据的命名实体。

变量

变量是用来存储数据的。

一个变量的名称必须以字母字符开头，并且不能包含空格或者保留字符。

一个变量可以是一个指定的类型，表示它所存储的数据类型。string 变量存储字符串值（"Welcome to W3CSchool.cc"），integer 变量存储数字值（103），date 变量存储日期值，等等。

变量使用 Dim 关键字声明，或通过使用类型（如果您想声明类型）声明，但是 ASP.NET 通常能自动确定数据类型。

实例

```
// Using the Dim keyword:
Dim greeting = "Welcome to W3CSchool.cc"
Dim counter = 103
Dim today = DateTime.Today

// Using data types:
Dim greeting As String = "Welcome to W3CSchool.cc"
Dim counter As Integer = 103
Dim today As DateTime = DateTime.Today
```

数据类型

下面列出了常用的数据类型：

类型	描述	实例
integer	整数（全数字）	103, 12, 5168
double	64 位浮点数	3.14, 3.4e38
decimal	十进制数字（高精度）	1037.196543
boolean	布尔值	true, false
string	字符串	"Hello W3CSchool.cc", "John"

运算符

运算符告诉 ASP.NET 在表达式中执行什么样的命令。

VB 语言支持多种运算符。下面列出了常用的运算符：

运算符	描述	实例
=	给一个变量赋值。	`i=6`
+ - * /	加上一个值或者一个变量。 减去一个值或者一个变量。 乘以一个值或者一个变量。 除以一个值或者一个变量。	`i=5+5` `i=5-5` `i=5*5` `i=5/5`
+= -=	变量递增。 变量递减。	`i += 1` `i -= 1`
=	相等。如果值相等则返回 true。	`if i=10`
<>	不等。如果值不等则返回 true。	`if <>10`
< > <= >=	小于。 大于。 小于等于。 大于等于。	`if i<10` `if i>10` `if i<=10` `if i>=10`
&	连接字符串（一系列互相关联的事物）。	`"w3" & "schools"`
.	点号。分隔对象和方法。	`DateTime.Hour`
()	圆括号。将值进行分组。	`(i+5)`
()	圆括号。传递参数。	`x=Add(i,5)`
()	圆括号。访问数组或者集合的值。	`name(3)`
Not	非。真/假取反。	`if Not ready`
And OR	逻辑与。 逻辑或。	`if ready And clear` `if ready Or clear`
AndAlso orElse	扩展的逻辑与。 扩展的逻辑或。	`if ready AndAlso clear` `if ready OrElse clear`

转换数据类型

从一种数据类型转换到另一种数据类型，有时候是很有用的。

最常见的例子是将字符串输入转换为另一种类型，如整数或者日期。

一般规则下，都是将用户输入看做字符串处理，即使用户输入了数字。因此数值输入必须被转换成数字，然后才能将其用于计算。

下面列出了常用的转换方法：

方法	描述	实例
AsInt() IsInt()	转换字符串为整数。	if myString.IsInt() then myInt=myString.AsInt() end if
AsFloat() IsFloat()	转换字符串为浮点数。	if myString.IsFloat() then myFloat=myString.AsFloat() end if
AsDecimal() IsDecimal()	转换字符串为十进制数。	if myString.IsDecimal() then myDec=myString.AsDecimal() end if
AsDateTime() IsDateTime()	转换字符串为 ASP.NET DateTime 类型。	myString="10/10/2012" myDate=myString.AsDateTime()
AsBool() IsBool()	转换字符串为布尔值。	myString="True" myBool=myString.AsBool()
ToString()	转换任何数据类型为字符串。	myInt=1234 myString=myInt.ToString()

ASP.NET Razor - VB 循环和数组

语句在循环中会被重复执行。

For 循环

如果您需要重复执行相同的语句，您可以设定一个循环。

如果您知道要循环的次数，您可以使用 **for** 循环。这种类型的循环在向上计数或向下计数时特别有用：

实例

```
<html>
<body>
@For i=10 To 21
@<p>Line #@i</p>
Next i
</body>
</html>
```

运行实例？

For Each 循环

如果您使用的是集合或者数组，您会经常用到 **for each** 循环。

集合是一组相似的对象，for each 循环可以遍历集合直到完成。

下面的实例中，遍历 ASP.NET Request.ServerVariables 集合。

实例

```
<html>
<body>
<ul>
@For Each x In Request.ServerVariables
@<li>@x</li>
Next x
</ul>
</body>
</html>
```

运行实例？

While 循环

while 循环是一个通用的循环。

while 循环以 while 关键字开始，后面紧跟着括号，您可以在括号里规定循环将持续多久，然后是重复执行的代码块。

while 循环通常会设定一个递增或者递减的变量用来计数。

下面的实例中，+= 运算符在每执行一次循环时给变量 i 的值加 1。

实例

```
<html>
<body>
@Code
Dim i=0
Do While i<5
i += 1
@<p>Line #@i</p>
Loop
End Code
</body>
</html>
```

运行实例？

数组

当您要存储多个相似变量但又不想为每个变量都创建一个独立的变量时，可以使用数组来存储：

实例

```
@Code
Dim members As String()={"Jani","Hege","Kai","Jim"}
i=Array.IndexOf(members,"Kai")+1
len=members.Length
x=members(2-1)
end Code
<html>
<body>
<h3>Members</h3>
@For Each person In members
@<p>@person</p>
Next person
<p>The number of names in Members are @len</p>
<p>The person at position 2 is @x</p>
<p>Kai is now in position @i</p>
</body>
</html>
```

运行实例？

ASP.NET Razor - VB 逻辑条件

编程逻辑：根据条件执行代码。

If 条件

VB 允许根据条件执行代码。

使用 **if** 语句来判断条件。根据判断结果，if 语句返回 true 或者 false：

- if 语句开始一个代码块
- 条件写在 if 和 then 之间
- 如果条件为真，if ... then 和 end if 之间的代码被执行

实例

```
@Code
Dim price=50
End Code
<html>
<body>
@If price>30 Then
@<p>The price is too high.</p>
End If
</body>
</html>
```

运行实例？

Else 条件

if 语句可以包含 **else** 条件。

else 条件定义了当条件为假时被执行的代码。

实例

```
@Code
Dim price=20
End Code
<html>
<body>
@if price>30 then
@<p>The price is too high.</p>
Else
@<p>The price is OK.</p>
End If
</body>
</htmlV>
```

运行实例？

注释：在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。else 条件覆盖了除 if 条件之外的"其他所有情况"。

ElseIf 条件

多个条件判断可以使用 **elseif** 条件：

实例

```
@Code
Dim price=25
End Code
<html>
<body>
@If price>=30 Then
@<p>The price is high.</p>
ElseIf price>20 And price<30
@<p>The price is OK.</p>
Else
@<p>The price is low.</p>
End If
</body>
</html>
```

运行实例？

在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。

如果第一个条件不为真且第二个条件为真，elseif 块的代码将会被执行。

elseif 条件的数量不受限制。

如果 if 和 elseif 条件都不为真，最后的 else 块（不带条件）覆盖了"其他所有情况"。

Select 条件

select 块可以用来测试一些单独的条件：

实例

```
@Code
Dim weekday=DateTime.Now.DayOfWeek
Dim day=weekday.ToString()
Dim message=""
End Code
<html>
<body>
@Select Case day
Case "Monday"
message="This is the first weekday."
Case "Thursday"
message="Only one day before weekend."
Case "Friday"
message="Tomorrow is weekend!"
Case Else
message="Today is " & day
End Select
<p>@message</p>
</body>
</html>
```

运行实例？

"Select Case" 后面紧跟着测试值（day）。每个单独的测试条件都有一个 case 值和任意数量的代码行。如果测试值与 case 值相匹配，相应的代码行被执行。

select 块有一个默认的情况（Case Else），当所有的指定的情况都不匹配时，它覆盖了"其他所有情况"。

MVC 教程

ASP.NET MVC 教程

ASP.NET 是一个使用 HTML、CSS、JavaScript 和服务器脚本创建网页和网站的开发框架。

ASP.NET 支持三种不同的开发模式：
Web Pages（Web 页面）、MVC（Model View Controller 模型-视图-控制器）、Web Forms（Web 窗体）。

本教程介绍 **MVC**。

Web Pages
MVC
Web Forms

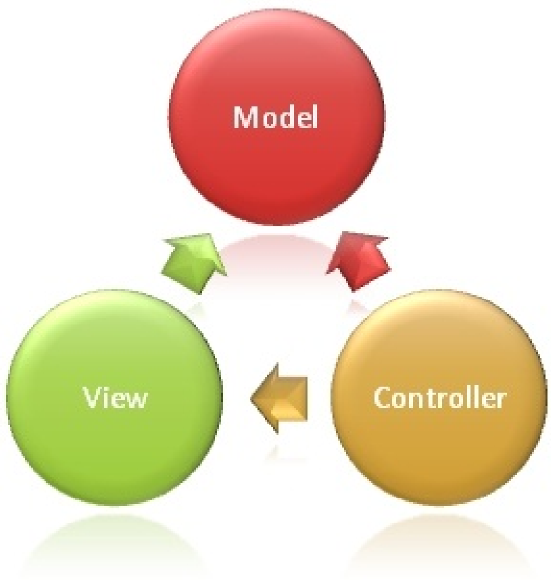
MVC 编程模式

MVC 是三种 ASP.NET 编程模式中的一种。

MVC 是一种使用 MVC（Model View Controller 模型-视图-控制器）设计创建 Web 应用程序的模式：

- Model（模型）表示应用程序核心（比如数据库记录列表）。
- View（视图）显示数据（数据库记录）。
- Controller（控制器）处理输入（写入数据库记录）。

MVC 模式同时提供了对 HTML、CSS 和 JavaScript 的完全控制。



MVC 模式定义 Web 应用程序 带有三个逻辑层： 业务层（模型逻辑） 显示层（视图逻辑） 输入控制（控制器逻辑） |

Model（模型）是应用程序中用于处理应用程序数据逻辑的部分。
通常模型对象负责在数据库中存取数据。

View（视图）是应用程序中处理数据显示的部分。
通常视图是依据模型数据创建的。

Controller（控制器）是应用程序中处理用户交互的部分。
通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

MVC 分层有助于管理复杂的应用程序，因为您可以在一个时间内专门关注一个方面。例如，您可以在不依赖业务逻辑的情况下专注于视图设计。同时也让应用程序的测试更加容易。

MVC 分层同时也简化了分组开发。不同的开发人员可同时开发视图、控制器逻辑和业务逻辑。

Web Forms 对比 MVC

MVC 编程模式是对传统 ASP.NET（Web Forms）的一种轻量级的替代方案。它是轻量级的、可测试性高的框架，同时整合了所有已有的 ASP.NET 特性，比如母版页、安全性和认证。

Visual Studio Express 2012/2010

Visual Studio Express 是 Microsoft Visual Studio 的免费版本。

Visual Studio Express 是为 MVC（和 Web Forms）量身定制的开发工具。

Visual Studio Express 包含：

- MVC 和 Web Forms
- 拖拽 Web 控件和 Web 组件
- Web 服务器语言（Razor 使用 VB 或者 C#）
- Web 服务器（IIS Express）
- 数据库服务器（SQL Server Compact）
- 完整的 Web 开发框架（ASP.NET）

如果您已经安装了 Visual Studio Express，您将从本教程中学到更多。

如果您想安装 Visual Studio Express，请点击下列链接中的一个：

Visual Web Developer 2012（Windows 7 或者 Windows 8）

Visual Web Developer 2010（Windows Vista 或者 XP）

> 在您首次安装完 Visual Studio Express 之后，您可以通过再次运行安装程序来安装补丁和服务包，只需要再次点击链接即可。 |

ASP.NET MVC 参考手册

在本教程的最后，我们提供了完整的 ASP.NET MVC 参考手册供您查阅。

ASP.NET MVC - Internet 应用程序

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 1 部分：创建应用程序。

我们将构建什么

我们将构建一个支持添加、编辑、删除和列出数据库存储信息的 Internet 应用程序。

我们将做什么

Visual Web Developer 提供了构建 Web 应用程序的不同模板。

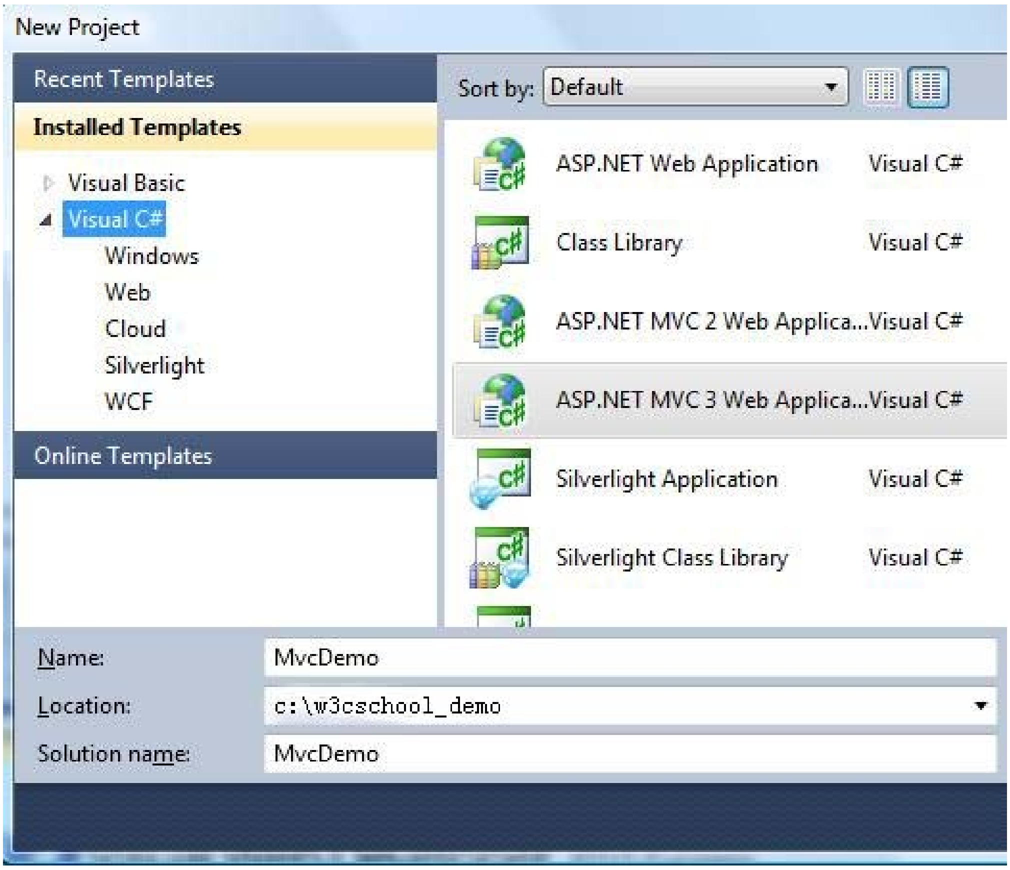
我们将使用 **Visual Web Developer** 来创建一个带 **HTML5** 标记的空的 MVC Internet 应用程序。

当这个空白的 Internet 应用程序被创建之后，我们将逐步向该应用添加代码，直到全部完成。我们将使用 **C#** 作为编程语言，并使用最新的 **Razor** 服务器代码标记。

沿着这个思路，我们将讲解这个应用程序的内容、代码和所有组件。

创建 Web 应用程序

如果您已经安装了 Visual Web Developer，请启动 Visual Web Developer 并选择 **New Project** 来新建项目。否则您就只能通过阅读教程来学习了。



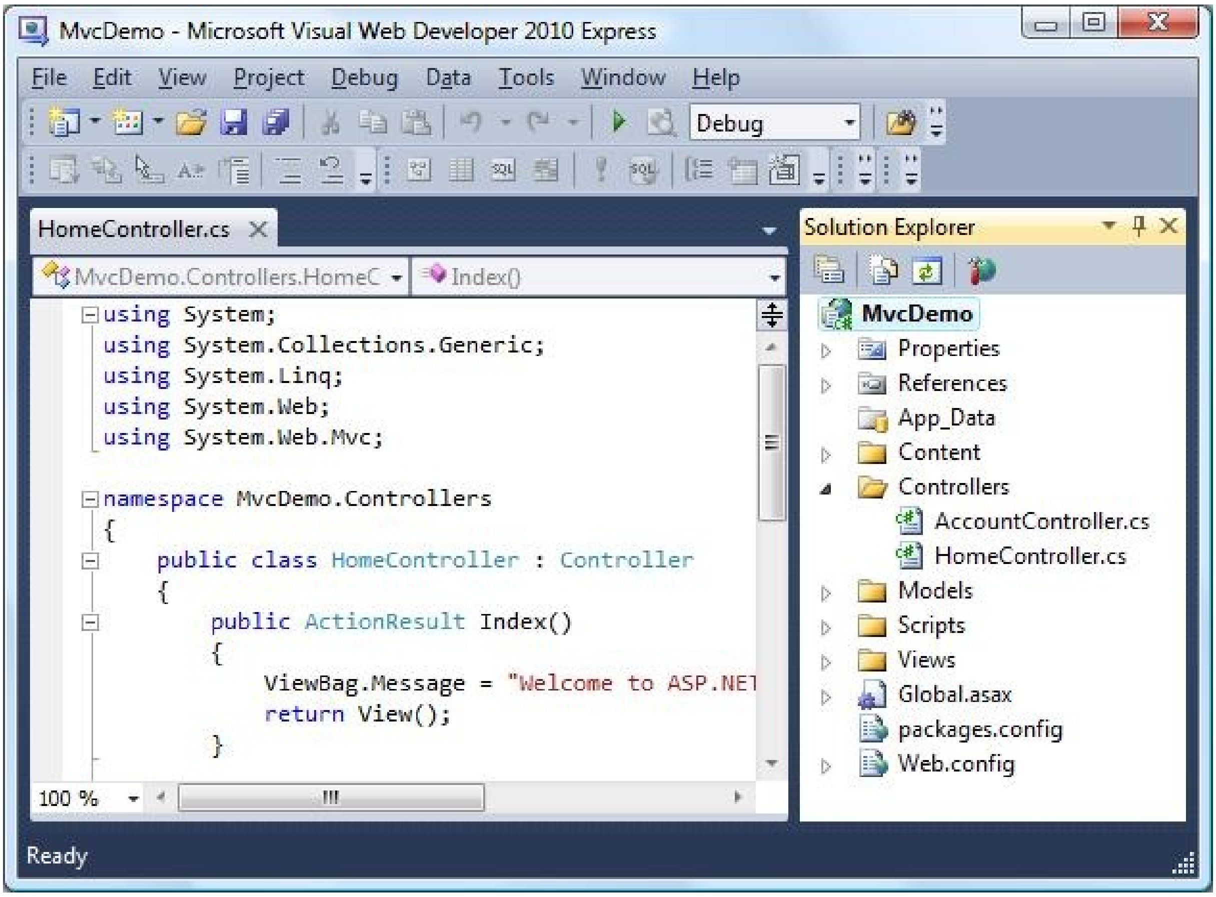
在 New Project 对话框中：

- 打开**Visual C#**模板
- 选择模板 **ASP.NET MVC 3 Web Application**
- 设置项目名称为 **MvcDemo**
- 设置磁盘位置，比如 **c:\w3cschool_demo**
- 点击 **OK**

当 New Project 对话框打开时：

- 选择 **Internet Application** 模板
- 选择 **Razor Engine**（Razor 引擎）
- 选择 **HTML5 Markup**（HTML5 标记）
- 点击 **OK**

Visual Studio Express 将创建一个如下所示的类似项目：



我们将在本教程的下一章中探究有关文件和文件夹的内容。

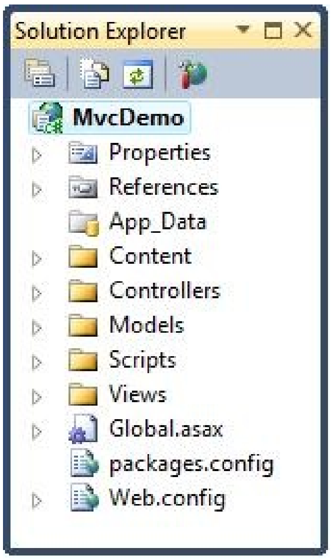
ASP.NET MVC - 应用程序文件夹

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 2 部分：探究应用程序文件夹。

MVC 文件夹

一个典型的 ASP.NET MVC Web 应用程序的文件夹内容如下所示：



应用程序信息

Properties
References

应用程序文件夹

App_Data 文件夹
Content 文件夹
Controllers 文件夹
Models 文件夹
Scripts 文件夹
Views 文件夹

配置文件

Global.asax
packages.config
Web.config

所有的 MVC 应用程序的文件夹名称都是相同的。MVC 框架是基于默认的命名。控制器写在 Controllers 文件夹中，视图写在 Views 文件夹中，模型写在 Models 文件夹中。您不必再应用程序代码中使用文件夹名称。

标准化的命名减少了代码量，同时有利于开发人员对 MVC 项目的理解。

下面是对每个文件夹内容的简短概述：

App_Data 文件夹

App_Data 文件夹用于存储应用程序数据。

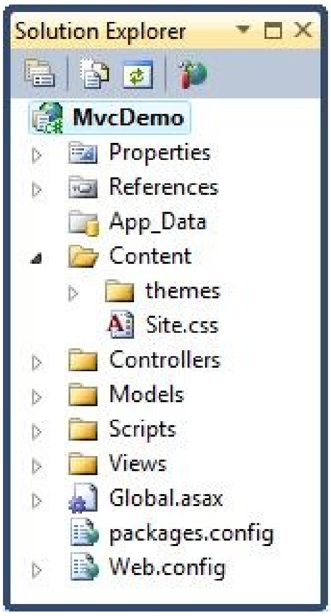
我们将在本教程后面的章节中介绍添加 SQL 数据库到 App_Data 文件夹。

Content 文件夹

Content 文件夹用于存放静态文件，比如样式表（CSS 文件）、图标和图像。

Visual Web Developer 会自动添加一个 **themes** 文件夹到 Content 文件夹中。themes 文件夹存放 jQuery 样式和图片。在项目中，您可以删除这个 themes 文件夹。

Visual Web Developer 同时也会添加一个标准的样式表文件到项目中：即 content 文件夹中的 **Site.css** 文件。这个样式表文件是您想要改变应用程序样式时需要编辑的文件。



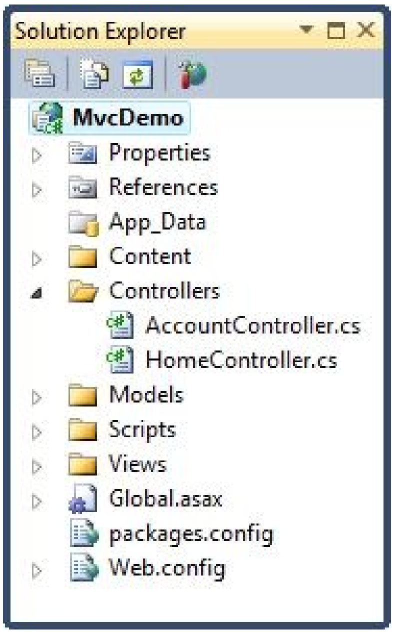
我们将在本教程的下一章中编辑这个样式表文件（Site.css）。

Controllers 文件夹

Controllers 文件夹包含负责处理用户输入和相应的控制器类。

MVC 要求所有控制器文件的名称以 "Controller" 结尾。

Visual Web Developer 已经创建好一个 Home 控制器（用于 Home 页面和 About 页面）和一个 Account 控制器（用于 Login 页面）：



我们将在本教程后面的章节中创建更多的控制器。

Models 文件夹

Models 文件夹包含表示应用程序模型的类。模型控制并操作应用程序的数据。

我们将在本教程后面的章节中创建模型（类）。

Views 文件夹

Views 文件夹用于存储与应用程序的显示相关的 HTML 文件（用户界面）。

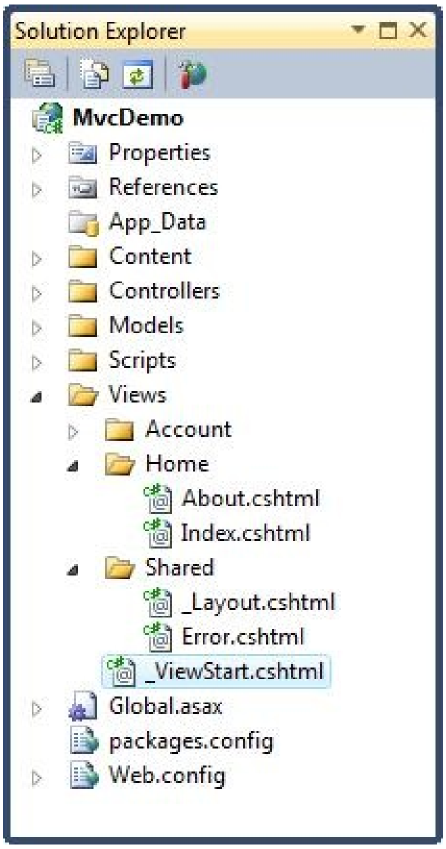
Views 文件夹中包含每个控制器对应的一个文件夹。

在 Views 文件夹中，Visual Web Developer 已经创建了一个 Account 文件夹、一个 Home 文件夹、一个 Shared 文件夹。

Account 文件夹包含用于用户账号注册和登录的页面。

Home 文件夹用于存储诸如 home 页和 about 页之类的应用程序页面。

Shared 文件夹用于存储控制器间分享的视图（母版页和布局页）。

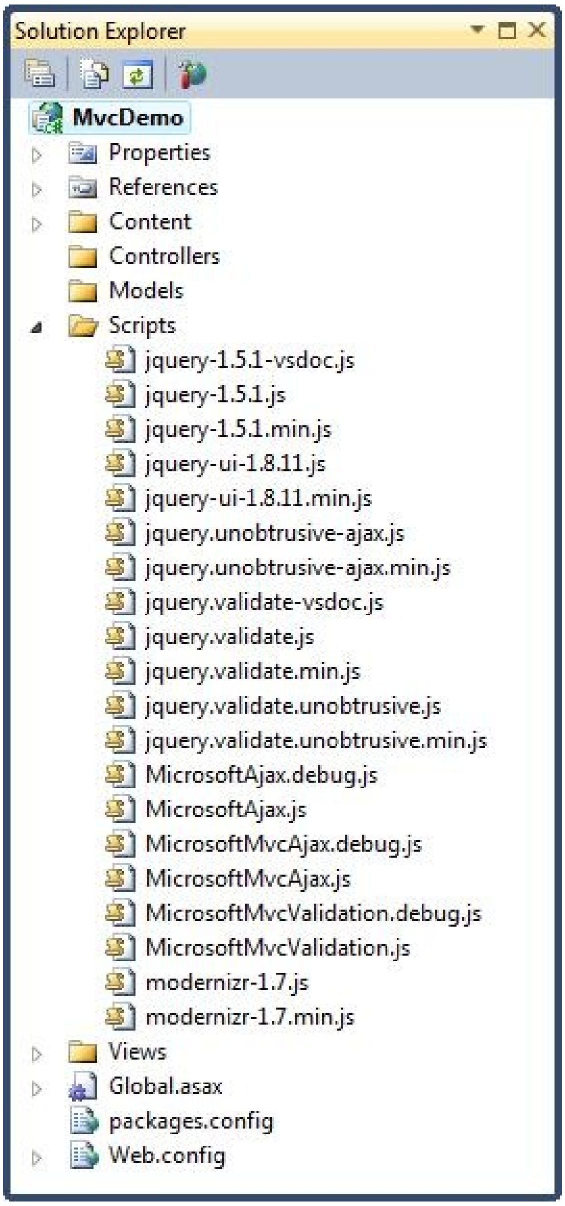


我们将在本教程的下一章中编辑这些布局文件。

Scripts 文件夹

Scripts 文件夹存储应用程序的 JavaScript 文件。

默认情况下，Visual Web Developer 在这个文件夹中存放标准的 MVC、Ajax 和 jQuery 文件：



注释：名为 "modernizr" 的文件时用于在应用程序中支持 HTML5 和 CSS3 的 JavaScript 文件。

ASP.NET MVC - 样式和布局

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 3 部分：添加样式和统一的外观（布局）。

添加布局

文件 _Layout.cshtml 表示应用程序中每个页面的布局。它位于 Views 文件夹中的 Shared 文件夹。

打开文件 _Layout.cshtml，把内容替换成：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>@ViewBag.Title</title>
<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"></script>
<script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")"></script>
</head>
<body>
<ul id="menu">
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("Movies", "Index", "Movies")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
</ul>
<section id="main">
@RenderBody()
<p>Copyright W3CSchool 2012\. All Rights Reserved.</p>
</section>
</body>
</html>
```

HTML 帮助器

在上面的代码中，HTML 帮助器用于修改 HTML 输出：

```
@Url.Content() - URL 内容将在此处插入。

@Html.ActionLink() - HTML 链接将在此处插入。
```

在本教程后面的章节中，您将学到更多关于 HTML 帮助器的知识。

Razor 语法

在上面的代码中，红色标记的代码是使用 Razor 标记的 C#。

```
@ViewBag.Title - 页面标题将在此处插入。

@RenderBody() - 页面内容将在此处呈现。
```

您可以在我们的 Razor 教程中学习关于 C# 和 VB（Visual Basic）的 Razor 标记的知识。

添加样式

应用程序的样式表是 Site.css，位于 Content 文件夹中。

打开文件 Site.css，把内容替换成：

```
body
{
font: "Trebuchet MS", Verdana, sans-serif;
background-color: #5c87b2;
color: #696969;
}
h1
{
border-bottom: 3px solid #cc9900;
font: Georgia, serif;
color: #996600;
}
#main
{
padding: 20px;
background-color: #ffffff;
border-radius: 0 4px 4px 4px;
}
a
{
color: #034af3;
}
/* Menu Styles ------------------------------*/
ul#menu
{
padding: 0px;
position: relative;
margin: 0;
}
ul#menu li
{
display: inline;
}
ul#menu li a
{
background-color: #e8eef4;
padding: 10px 20px;
text-decoration: none;
line-height: 2.8em;
/*CSS3 properties*/
border-radius: 4px 4px 0 0;
}
ul#menu li a:hover
{
background-color: #ffffff;
}
/* Forms Styles ------------------------------*/
fieldset
{
padding-left: 12px;
}
```

```
fieldset label
{
display: block;
padding: 4px;
}
input[type="text"], input[type="password"]
{
width: 300px;
}
input[type="submit"]
{
padding: 4px;
}
/* Data Styles ------------------------------*/
table.data
{
background-color:#ffffff;
border:1px solid #c3c3c3;
border-collapse:collapse;
width:100%;
}
table.data th
{
background-color:#e8eef4;
border:1px solid #c3c3c3;
padding:3px;
}
table.data td
{
border:1px solid #c3c3c3;
padding:3px;
}
```

_ViewStart 文件

Shared 文件夹（位于 Views 文件夹内）中的 _ViewStart 文件包含如下内容：

```
@{Layout = "~/Views/Shared/_Layout.cshtml";}
```

这段代码被自动添加到由应用程序显示的所有视图。

如果您删除了这个文件，则必须向所有视图中添加这行代码。

在本教程后面的章节中，您将学到更多关于视图的知识。

ASP.NET MVC - 控制器

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

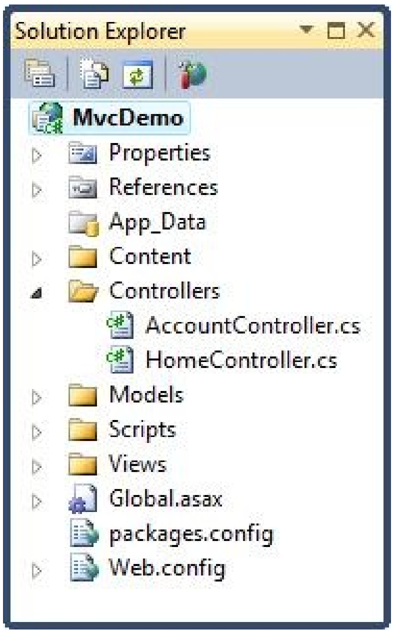
第 4 部分：添加控制器。

Controllers 文件夹

Controllers 文件夹包含负责处理用户输入和响应的控制类。

MVC 要求所有控制器文件的名称以 "Controller" 结尾。

在我们的实例中，Visual Web Developer 已经创建好了一下文件： **HomeController.cs**（用于 Home 页面和 About 页面）和**AccountController.cs** （用于登录页面）：



Web 服务器通常会将进入的 URL 请求直接映射到服务器上的磁盘文件。例如：URL 请求 "http://www.w3cschool.cc/index.php" 将直接映射到服务器根目录上的文件 "index.php"。

MVC 框架的映射方式有所不同。MVC 将 URL 映射到方法。这些方法在类中被称为"控制器"。

控制器负责处理进入的请求，处理输入，保存数据，并把响应发送回客户端。

Home 控制器

在我们应用程序中的控制器文件**HomeController.cs**，定义了两个控件 **Index** 和 **About**。

把 HomeController.cs 文件的内容替换成：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcDemo.Controllers
{
public class HomeController : Controller
{
public ActionResult Index()
{return View();}

public ActionResult About()
{return View();}
}
}
```

Controller 视图

Views 文件夹中的文件 **Index.cshtml** 和 **About.cshtml** 定义了控制器中的 ActionResult 视图 Index() 和 About()。

ASP.NET MVC - 视图

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 5 部分：添加用于显示应用程序的视图。

Views 文件夹

Views 文件夹存储的是与应用程序显示（用户界面）相关的文件（HTML 文件）。根据所采用的语言内容，这些文件可能扩展名可能是 html、asp、aspx、cshtml 和 vbhtml。

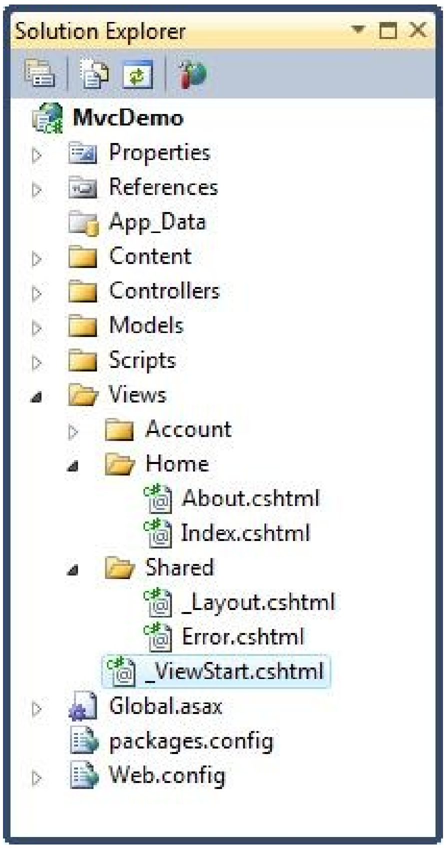
Views 文件夹中包含每个控制器对应的一个文件夹。

在 Views 文件夹中，Visual Web Developer 已经创建了一个 Account 文件夹、一个 Home 文件夹、一个 Shared 文件夹。

Account 文件夹包含用于用户账号注册和登录的页面。

Home 文件夹用于存储诸如 home 页和 about 页之类的应用程序页面。

Shared 文件夹用于存储控制器间分享的视图（母版页和布局页）。



ASP.NET 文件类型

在 Views 文件夹中可以看到以下 HTML 文件类型：

文件类型	扩展名
纯 HTML	.htm or .html
经典 ASP	.asp
经典 ASP.NET	.aspx
ASP.NET Razor C#	.cshtml
ASP.NET Razor VB	.vbhtml

Index 文件

文件 Index.cshtml 表示应用程序的 Home 页面。它是应用程序的默认文件（首页文件）。

在文件中写入以下内容：

```
@{ViewBag.Title = "Home Page";}

<h1>Welcome to W3CSchool.cc</h1>

<p>Put Home Page content here</p>
```

About 文件

文件 About.cshtml 表示应用程序的 About 页面。

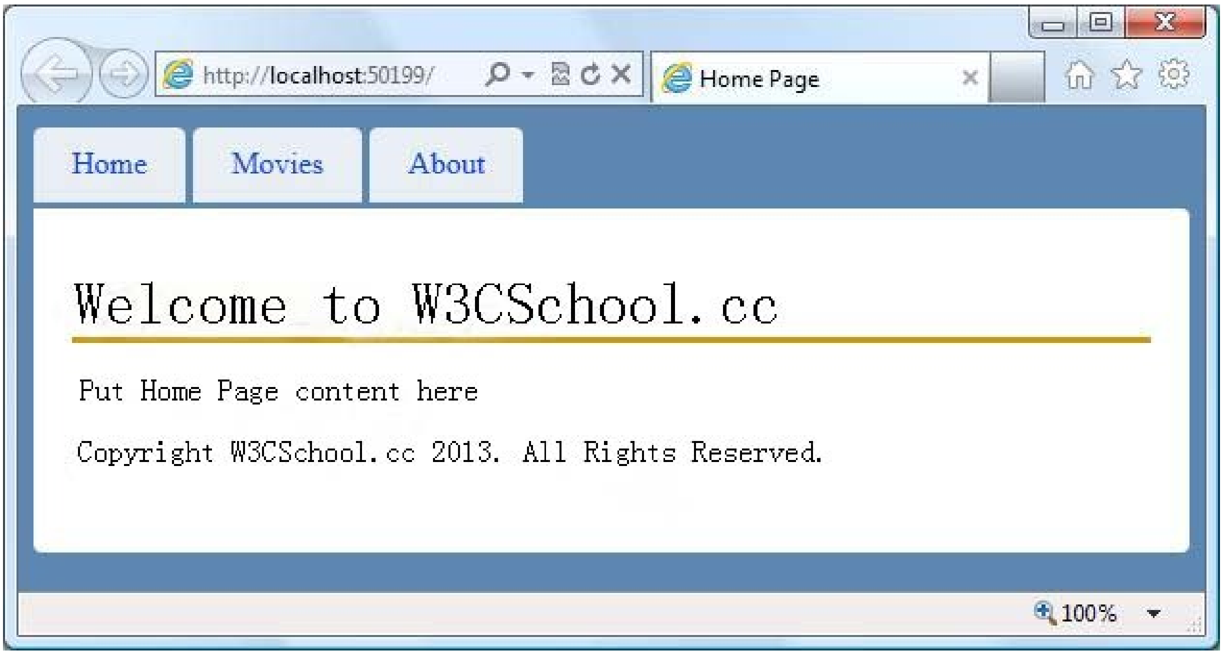
在文件中写入以下内容：

```
@{ViewBag.Title = "About Us";}

<h1>About Us</h1>

<p>Put About Us content here</p>
```

运行应用程序

选择 Debug，从 Visual Web Developer 菜单中启动调试 Start Debugging（或者按 F5）。

您的应用程序将显示如下：



点击 "Home" 标签页和 "About" 标签页，看看它是如何运作的。

祝贺您

祝贺您。您已经创建好了您的第一个 MVC 应用程序。

注释：您暂时还不能点击 "Movies" 标签页。我们将在本教程的后面章节中为 "Movies" 标签页添加代码。

ASP.NET MVC - SQL 数据库

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 6 部分：添加数据库。

创建数据库

Visual Web Developer 带有名为 SQL Server Compact 免费的 SQL 数据库。

本教程所需的这个数据库可以通过以下几个简单的步骤来创建：

- 右击 **Solution Explorer** 窗口中的 **App_Data** 文件夹
- 选择 **Add, New Item**
- 选择 **SQL Server Compact Local Database** *
- 将数据库命名为 **Movies.sdf**
- 点击 **Add** 按钮

* 如果选项中没有 SQL Server Compact Local Database，则说明您尚未在计算机上安装 SQL Server Compac。请通过以下链接进行安装：SQL Server Compact

Visual Web Developer 会自动在 App_Data 文件夹中创建该数据库。

注释：在本教程中，需要您掌握一些关于 SQL 数据库的基础知识。如果您想先学习这个主题，请访问我们的 SQL 教程。

添加数据库表

双击 **App_Data** 文件夹中的 **Movies.sdf** 文件，将打开 **Database Explorer** 窗口。

如需在数据库中创建一个新的表，请右击 **Tables** 文件夹，然后选择 **Create Table**。

创建如下的列：

列	类型	是否允许为 **Null**
ID	int (primary key)	No
Title	nvarchar(100)	No
Director	nvarchar(100)	No
Date	datetime	No

对列的解释：

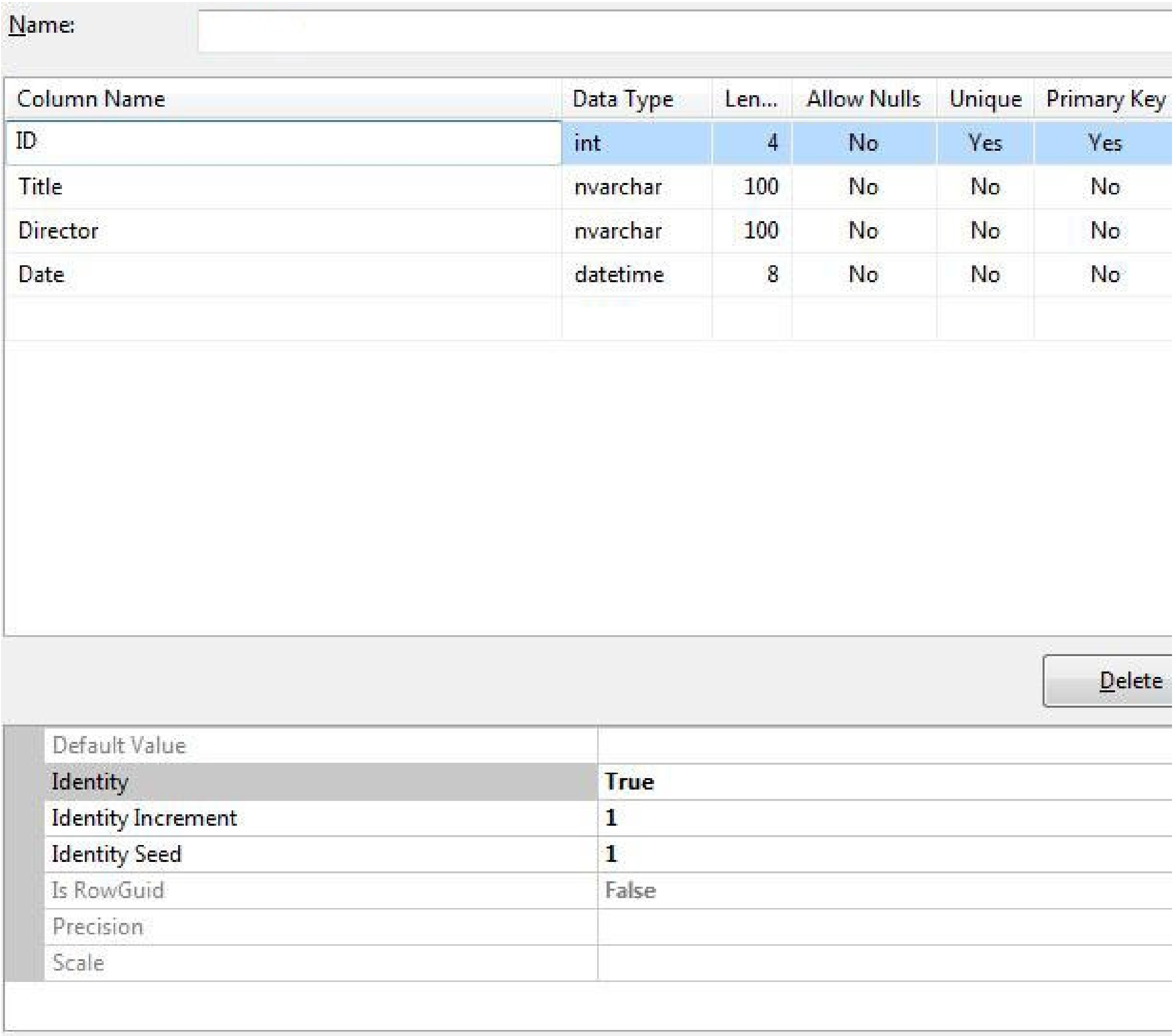
ID 是用于标识表中每条记录的整数（全数字）。

Title 是 100 个字符长度的文本列，用于存储影片的名称。

Director 是 100 个字符长度的文本列，用于存储导演的名字。

Date 是日期列，用于存储影片的发布日期。

在创建好上述列之后，您必须将 ID 列设置为表的主键（记录标识符）。要做到这点，请点击列名（ID），并选择 **Primary Key**。在 **Column Properties** 窗口中，设置 **Identity** 属性为 **True**：



当您创建好表列后，保存表并命名为 **MovieDBs**。

注释：

我们特意把表命名为 "MovieDBs"（以 s 结尾）。在下一章中，您将看到用于数据模型的 "MovieDB"。这看起来有点奇怪，不过这种命名惯例能确保控制器连接上数据库表，您必须这么使用。

添加数据库记录

您可以使用 Visual Web Developer 向 movie 数据库中添加一些测试记录。

双击 **App_Data** 文件夹中的 **Movies.sdf** 文件。

右击 Database Explorer 窗口中的 **MovieDBs** 表，并选择 **Show Table Data**。

添加一些记录：

ID	Title	Director	Date
1	Psycho	Alfred Hitchcock	01.01.1960
2	La Dolce Vita	Federico Fellini	01.01.1960

注释：ID 列会自动更新，您可以不用编辑它。

添加连接字符串

向您的 **Web.config** 文件中的 **<connectionStrings>** 元素添加如下元素：

ASP.NET MVC - 模型

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 7 部分：添加数据模型。

MVC 模型

MVC 模型包含了除纯视图和控制器逻辑以外的其他所有应用程序逻辑（业务逻辑、验证逻辑、数据访问逻辑）。

通过 MVC，模型可以控制并操作应用程序数据。

Models 文件夹

Models 文件夹包含表示应用程序模型的类。

Visual Web Developer 自动创建一个 **AccountModels.cs** 文件，该文件包含用于应用程序安全的模型。

AccountModels 包含 **LogOnModel**、**ChangePasswordModel** 和 **RegisterModel**。

添加数据库模型

本教程所需的数据库模型可以通过以下几个简单的步骤来创建：

- 在 **Solution Explorer**窗口中，右击 **Models** 文件夹，并选择 **Add** 和 **Class**。
- 将类命名为 **MovieDB.cs**，然后点击 **Add**。
- 编辑这个类：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;

namespace MvcDemo.Models
{
public class MovieDB
{
public int ID { get; set; }
public string Title { get; set; }
public string Director { get; set; }
public DateTime Date { get; set; }

}
public class MovieDBContext : DbContext
{
public DbSet<MovieDB> Movies { get; set; }
}
}
```

注释：

我们特意把模型命名为 "MovieDB"。在上一章中，您已经看到用于数据库表的 "MovieDBs"（以 s 结尾）。这看起来有点奇怪，不过这种命名惯例能确保模型连接上数据库表，您必须这么使用。

添加数据库控制器

本教程所需的数据库控制器可以通过以下几个简单的步骤来创建：

- 重建您的项目：选择 **Debug**，然后从菜单中选择 **Build MvcDemo**。
- 在 Solution Explorer（解决方案资源管理器）中，右击 **Controllers** 文件夹，选择 **Add** 和 **Controller**。
- 设置控制器名称为 **MoviesController**。
- 选择模板：**Controller with read/write actions and views, using Entity Framework**
- 选择模型类：**MovieDB (MvcDemo.Models)**
- 选择 data context 类：**MovieDBContext (MvcDemo.Models)**
- 选择视图 **Razor (CSHTML)**
- 点击 **Add**

Visual Web Developer 将创建以下文件：

- **Controllers** 文件夹中的 **MoviesController.cs** 文件
- **Views** 文件夹中的 **Movies** 文件夹

添加数据库视图

在 Movies 文件夹中，会自动创建以下文件：

- Create.cshtml
- Delete.cshtml
- Details.cshtml
- Edit.cshtml
- Index.cshtml

祝贺您

祝贺您。您已经向应用程序添加了您的第一个 MVC 数据模型。

现在您可以点击 "Movies" 标签页了。

ASP.NET MVC - 安全

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

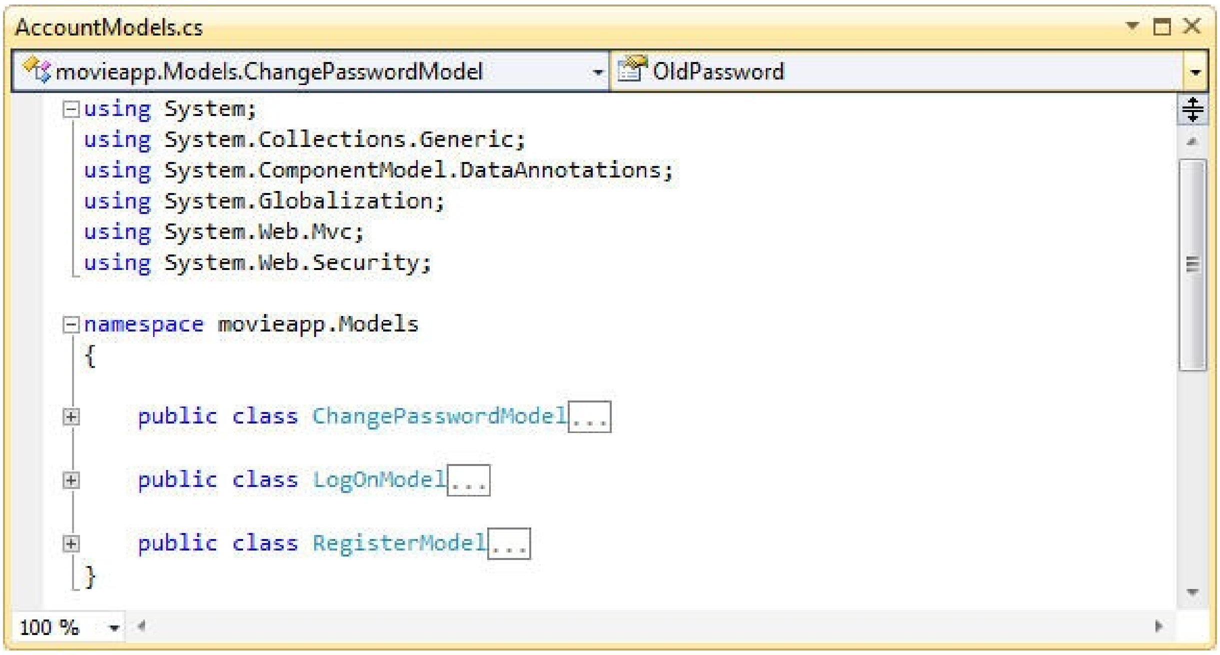
第 8 部分：添加安全。

MVC 应用程序安全

Models 文件夹包含表示应用程序模型的类。

Visual Web Developer 自动创建 **AccountModels.cs** 文件，该文件包含用于应用程序认证的模型。

AccountModels 包含 **LogOnModel**、**ChangePasswordModel** 和 **RegisterModel**：



Change Password 模型

```
public class ChangePasswordModel
{

[Required]
[DataType(DataType.Password)]
[Display(Name = "Current password")]
public string OldPassword { get; set; }

[Required]
[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", Minimu
[DataType(DataType.Password)]
[Display(Name = "New password")]
public string NewPassword { get; set; }

[DataType(DataType.Password)]
[Display(Name = "Confirm new password")]
[Compare("NewPassword", ErrorMessage = "The new password and confirmation password do not
public string ConfirmPassword { get; set; }

}
```

Logon 模型

```
public class LogOnModel
{

[Required]
[Display(Name = "User name")]
public string UserName { get; set; }

[Required]
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }

[Display(Name = "Remember me?")]
public bool RememberMe { get; set; }

}
```

Register 模型

```
public class RegisterModel
{

[Required]
[Display(Name = "User name")]
public string UserName { get; set; }

[Required]
[DataType(DataType.EmailAddress)]
[Display(Name = "Email address")]
public string Email { get; set; }

[Required]
[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", Minimu
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }

[DataType(DataType.Password)]
[Display(Name = "Confirm password")]
[Compare("Password", ErrorMessage = "The password and confirmation password do not match.
public string ConfirmPassword { get; set; }

}
```

ASP.NET MVC - HTML 帮助器

HTML 帮助器用于修改 HTML 输出。

HTML 帮助器

通过 MVC，HTML 帮助器类似于传统的 ASP.NET Web Form 控件。

就像 ASP.NET 中的 Web Form 控件，HTML 帮助器用于修改 HTML。但是 HTML 帮助器是更轻量级的。与 Web Form 控件不同，HTML 帮助器没有事件模型和视图状态。

在大多数情况下，HTML 帮助器仅仅是一个返回字符串的方法。

通过 MVC，您可以创建您自己的帮助器，或者直接使用内建的 HTML 帮助器。

标准的 HTML 帮助器

MVC 包含了大多数常用的 HTML 元素类型的标准帮助器，比如 HTML 链接和 HTML 表单元素。

HTML 链接

呈现 HTML 链接的最简单的方法是使用 HTML.ActionLink() 帮助器。

通过 MVC，Html.ActionLink() 不连接到视图。它创建一个连接到控制器操作。

Razor 语法：

```
@Html.ActionLink("About this Website", "About")
```

ASP 语法：

```
<%=Html.ActionLink("About this Website", "About")%>
```

第一个参数是链接文本，第二个参数是控制器操作的名称。

上面的 Html.ActionLink() 帮助器，输出以下的 HTML：

```
<a href="/Home/About">About this Website</a>
```

Html.ActionLink() 帮助器的一些属性：

属性	描述
.linkText	URL 文本（标签），定位点元素的内部文本。
.actionName	操作（action）的名称。
.routeValues	传递给操作（action)的值，是一个包含路由参数的对象。
.controllerName	控制器的名称。
.htmlAttributes	URL 的属性设置，是一个包含要为该元素设置的 HTML 特性的对象。
.protocol	URL 协议，如 "http" 或 "https"。
.hostname	URL 的主机名。
.fragment	URL 片段名称（定位点名称）。

注释：您可以向控制器操作传递值。例如，您可以向数据库 Edit 操作传递数据库记录的 id：

Razor 语法 C#：

```
@Html.ActionLink("Edit Record", "Edit", new {Id=3})
```

Razor 语法 VB：

```
@Html.ActionLink("Edit Record", "Edit", New With{.Id=3})
```

上面的 Html.ActionLink() 帮助器，输出以下的 HTML：

```
<a href="/Home/Edit/3">Edit Record</a>
```

HTML 表单元素

以下 HTML 帮助器可用于呈现（修改和输出）HTML 表单元素：

- BeginForm()
- EndForm()
- TextArea()
- TextBox()
- CheckBox()
- RadioButton()
- ListBox()
- DropDownList()

- Hidden()
- Password()

ASP.NET 语法 C#：

```
<%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try ag
<% using (Html.BeginForm()){%>
<p>
<label for="FirstName">First Name:</label>
<%= Html.TextBox("FirstName") %>
<%= Html.ValidationMessage("FirstName", "*") %>
</p>
<p>
<label for="LastName">Last Name:</label>
<%= Html.TextBox("LastName") %>
<%= Html.ValidationMessage("LastName", "*") %>
</p>
<p>
<label for="Password">Password:</label>
<%= Html.Password("Password") %>
<%= Html.ValidationMessage("Password", "*") %>
</p>
<p>
<label for="Password">Confirm Password:</label>
<%= Html.Password("ConfirmPassword") %>
<%= Html.ValidationMessage("ConfirmPassword", "*") %>
</p>
<p>
<label for="Profile">Profile:</label>
<%= Html.TextArea("Profile", new {cols=60, rows=10})%>
</p>
<p>
<%= Html.CheckBox("ReceiveNewsletter") %>
<label for="ReceiveNewsletter" style="display:inline">Receive Newsletter?</label>
</p>
<p>
<input type="submit" value="Register" />
</p>
<%}%>
```

ASP.NET MVC - 发布网站

学习如何在不使用 Visual Web Developer 的情况下发布 MVC 应用程序。

在不使用 Visual Web Developer 的情况下发布您的应用程序

通过在 WebMatrix、Visual Web Developer 或 Visual Studio 中使用发布命令，可以发布一个 ASP.NET MVC 应用程序到远程服务器上。

此功能会复制所有您的应用程序文件、控制器、模型、图像以及用于 MVC、Web Pages、Razor、Helpers、SQL Server Compact（如果使用数据库）所有必需的 DLL 文件。

有时您不希望使用这些选项。或许您的主机提供商仅支持 FTP？或许您的网站基于经典 ASP？或许您希望亲自拷贝这些文件？又或许您希望使用 Front Page、Expression Web 等其他一些发布软件？

您会遇到问题吗？是的，会的。但是您有办法解决它。

要执行网站复制，您必须知道如何引用正确的文件，哪些 DLL 文件需要复制，并在何处存储它们。

请按照下列步骤操作：

1. 使用最新版本的 ASP.NET

在您继续操作之前，请确保您的主机运行的是最新版的 ASP.NET（4.0 或者 4.5）。

2. 复制 Web 文件夹

从您的开发计算机上复制您的网站（所有文件夹和内容）到远程主机（服务器）上的应用程序文件夹中。

如果您的 **App_Data** 文件夹中包含测试数据，请不要复制这个 App_Data 文件夹（详见下面的第 5 点）。

3. 复制 DLL 文件

在远程服务器上的应用程序根目录中创建 bin 文件夹。（如果您已经安装 Helpers，则 bin 文件夹已经存在）

复制下列文件夹中的所有文件：

C:Program Files (x86)Microsoft ASP.NETASP.NET Web Pagesv1.0Assemblies

C:Program Files (x86)Microsoft ASP.NETASP.NET MVC 3Assemblies

到您的远程服务器上的应用程序的 bin 文件夹中。

4. 复制 SQL Server Compact DLL 文件

如果您的应用程序使用了 SQL Server Compact 数据库（在 App_Data 文件夹中的一个 .sdf 文件），那么您必须复制 SQL Server Compact DLL 文件：

复制下列文件夹中的所有文件：

C:Program Files (x86)Microsoft SQL Server Compact Editionv4.0Private

到您的远程服务器上的应用程序的 bin 文件夹中。

创建（或者编辑）应用程序的 Web.config 文件：

实例 C\

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.data>
<DbProviderFactories>
<remove invariant="System.Data.SqlServerCe.4.0" />

<add invariant="System.Data.SqlServerCe.4.0"
name="Microsoft SQL Server Compact 4.0"
description=".NET Framework Data Provider for Microsoft SQL Server Compact" type="System.

</DbProviderFactories>
</system.data>
</configuration>
```

5. 复制 SQL Server Compact 数据

您的 App_Data 文件夹中有没有包含测试数据的 .sdf 文件？

您是否希望发布您的测试数据到远程服务器上？

大多数时候一般是不希望。

如果您一定要复制 SQL 数据文件（.sdf 文件），那么您应该删除数据库中的所有数据，然后从您的开发计算机上复制一个空的 .sdf 文件到服务器上。

就是这样。**GOOD LUCK**！

Web Forms 教程

ASP.NET Web Forms - 教程

ASP.NET 是一个使用 HTML、CSS、JavaScript 和服务器脚本创建网页和网站的开发框架。

ASP.NET 支持三种不同的开发模式：
Web Pages（Web 页面）、MVC（Model View Controller 模型-视图-控制器）、Web Forms（Web 窗体）：

本教程介绍 **Web Forms**。

Web Pages
MVC
Web Forms

从何入手？

多数开发人员学习一个新技术，是从查看运行实例开始的。

如果您想查看一个 Web Forms 运行实例，请查看以下的 ASP.NET Web Forms 演示。

什么是 **Web Forms**？

Web Forms 是三种创建 ASP.NET 网站和 Web 应用程序的编程模式中的一种。

其他两种编程模式是 Web Pages 和 MVC（Model View Controller 模型-视图-控制器）。

Web Forms 是最古老的 ASP.NET 编程模式，是整合了 HTML、服务器控件和服务器代码的事件驱动网页。

Web Forms 是在服务器上编译和执行的，再由服务器生成 HTML 显示为网页。

Web Forms 有数以百计的 Web 控件和 Web 组件用来创建带有数据访问的用户驱动网站。

Visual Studio Express 2012/2010

Visual Studio Express 是 Microsoft Visual Studio 的免费版本。

Visual Studio Express 是为 Web Forms（和 MVC）量身定制的开发工具。

Visual Studio Express 包含：

- MVC 和 Web Forms

- 拖拽 Web 控件和 Web 组件
- Web 服务器语言（Razor 使用 VB 或者 C#）
- Web 服务器（IIS Express）
- 数据库服务器（SQL Server Compact）
- 完整的 Web 开发框架（ASP.NET）

如果您已经安装了 Visual Studio Express，您将从本教程中学到更多。

如果您想安装 Visual Studio Express，请点击下列链接中的一个：

Visual Web Developer 2012（Windows 7 或者 Windows 8）

Visual Web Developer 2010（Windows Vista 或者 XP）

ASP.NET 参考手册

在本教程的最后，您将看到一套完整的 ASP.NET 参考手册，介绍了对象、组件、属性和方法。

ASP.NET 参考手册

ASP.NET Web Forms - HTML 页面

简单的 ASP.NET 页面看上去就像普通的 HTML 页面。

Hello W3CSchool.cc

在开始学习 ASP.NET 之前，我们先来构建一个简单的 HTML 页面，该页面将在浏览器中显示 "Hello W3CSchool.cc"：

```
<table bgcolor="yellow" border="1" width="100%">

<tbody>

<tr><td>

## Hello W3CSchool.cc!

</td></tr>

</tbody>

</table>
```

用 HTML 编写的 Hello W3CSchool.cc

下面的代码将以 HTML 页面的形式显示实例：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
</center>
</body>
</html>
```

如果您想亲自尝试一下，请保存上面的代码到一个名为 "**firstpage.htm**" 的文件中，并创建一个到该文件的链接：firstpage.htm。

用 ASP.NET 编写的 Hello W3CSchool.cc

转换 HTML 页面为 ASP.NET 页面最简单的方法是，直接复制一个 HTML 文件，并把新文件的扩展名改成 **.aspx** 。

下面的代码将以 ASP.NET 页面的形式显示实例：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
</center>
</body>
</html>
```

如果您想亲自尝试一下，请保存上面的代码到一个名为 **"firstpage.aspx"** 的文件中，并创建一个到该文件的链接：firstpage.aspx。

它是如何工作的？

从根本上讲，ASP.NET 页面与 HTML 是完全相同的。

HTML 页面的扩展名是 .htm。如果浏览器向服务器请求一个 HTML 页面，服务器可以不进行任何修改，就直接发送页面给浏览器。

ASP.NET 页面的扩展名是 .aspx。如果浏览器向服务器请求个 ASP.NET 页面，服务器在将结果发回给浏览器之前，需要先处理页面中的可执行代码。

上面的 ASP.NET 页面不包含任何可执行的代码，所以没有执行任何东西。在下面的实例中，我们将添加一些可执行的代码到页面中，以便演示静态 HTML 页面和动态 ASP 页面的不同之处。

经典 ASP

Active Server Pages (ASP) 已经流行很多年了。通过 ASP，可以在 HTML 页面中放置可执行代码。

之前的 ASP 版本（在 ASP.NET 之前）通常被称为经典 ASP。

ASP.NET 不完全兼容经典 ASP，但是只需要经过少量的修改，大部分经典 ASP 页面就可以作为 ASP.NET 页面良好地运行。

如果您想学习更多关于经典 ASP 的知识，请访问我们的 ASP 教程。

用经典 ASP 编写的动态页面

为了演示 ASP 是如何显示包含动态内容的页面，我们将向上面的实例中添加一些可执行的代码（红色字体标识）：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
<p><%Response.Write(now())%></p>
</center>
</body>
</html>
```

<% --%> 标签内的代码是在服务器上执行的。

Response.Write 是用来向 HTML 输出流中写东西的 ASP 代码。

Now() 是一个返回服务器当前日期和时间的函数。

如果您想亲自尝试一下，请保存上面的代码到一个名为 "**dynpage.asp**" 的文件中，并创建一个到该文件的链接：dynpage.asp。

用 ASP .NET 编写的动态页面

下面的代码将以 ASP.NET 页面的形式显示实例：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
<p><%Response.Write(now())%></p>
</center>
</body>
</html>
```

如果您想亲自尝试一下，请保存上面的代码到一个名为 "**dynpage.aspx**" 的文件中，并创建一个到该文件的链接：dynpage.aspx。

ASP.NET 对比经典 ASP

上面的实例无法演示 ASP.NET 与经典 ASP 之间任何的不同之处。

正如最后的两个实例中，您看不出 ASP 页面和 ASP.NET 页面两者之间的不同之处。

在下一章中，您将看到服务器控件是如何让 ASP.NET 比经典 ASP 更强大的。

ASP.NET Web Forms - 服务器控件

服务器控件是服务器可理解的标签。

经典 ASP 的局限性

下面列出的代码是从上一章中复制的：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
<p><%Response.Write(now())%></p>
</center>
</body>
</html>
```

上面的代码反映出经典 ASP 的局限性：代码块必须放置在您想要输出显示的位置。

通过经典 ASP，想要把可执行代码从 HTML 页面中分离出来是不可能的。这让页面变得难以阅读，也难以维护。

ASP.NET - 服务器控件

ASP.NET 通过服务器控件，已经解决了上述的"意大利面条式代码"问题。

服务器控件是服务器可理解的标签。

有三种类型的服务器控件：

- HTML 服务器控件 - 创痛的 HTML 标签
- Web 服务器控件 - 新的 ASP.NET 标签
- Validation 服务器控件 - 用于输入验证

ASP.NET - HTML 服务器控件

HTML 服务器控件是服务器可理解的 HTML 标签。

ASP.NET 文件中的 HTML 元素，默认是作为文本进行处理的。要想让这些元素可编程，需向 HTML 元素中添加 runat="server" 属性。这个属性表示，该元素将被作为服务器控件进行处理。同时需要添加 id 属性来标识服务器控件。id 引用可用于操作运行时的服务器控件。

注释：所有 HTML 服务器控件必须位于带有 runat="server" 属性的 <form> 标签内。runat="server" 属性表明了该表单必须在服务器上进行处理。同时也表明了包含在它内部的控件可被服务器脚本访问。

在下面的实例中，我们在 .aspx 文件中声明了一个 HtmlAnchor 服务器控件。然后我们在一个事件句柄（事件句柄是一种针对给定事件执行代码的子例程）中操作 HtmlAnchor 控件的 HRef 属性。Page_Load 事件是 ASP.NET 可理解的多种事件中的一种：

```
<script runat="server">
Sub Page_Load
link1.HRef="http://www.w3cschool.cc"
End Sub
</script>

<html>
<body>

<form runat="server">
<a id="link1" runat="server">Visit W3CSchool.cc!</a>
</form>

</body>
</html>
```

可执行代码本身已经被移到 HTML 之外了。

ASP.NET - Web 服务器控件

Web 服务器控件是服务器可理解的特殊 ASP.NET 标签。

就像 HTML 服务器控件，Web 服务器控件也是在服务器上创建的，它们同样需要 runat="server" 属性才能生效。然而，Web 服务器控件没有必要映射任何已存在的 HTML 元素，它们可以表示更复杂的元素。

创建 Web 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

在下面的实例中，我们在 .aspx 文件中声明了一个 Button 服务器控件。然后我们为 Click 事件创建一个事件句柄，用来改变按钮上的文本：

```
<script runat="server">
Sub submit(Source As Object, e As EventArgs)
button1.Text="You clicked me!"
End Sub
</script>

<html>
<body>

<form runat="server">
<asp:Button id="button1" Text="Click me!"
runat="server" OnClick="submit"/>
</form>

</body>
</html>
```

ASP.NET - Validation 服务器控件

Validation 服务器控件是用来验证用户输入的。如果用户输入没有通过验证，将显示一条错误消息给用户。

每种 validation 控件执行一种指定类型的验证（比如验证摸个指定的值或者某个范围的值）。

在默认情况下，当 Button、ImageButton、LinkButton 控件被点击时，会执行页面验证。您可以设置 CausesValidation 为 false ，来阻止按钮控件被点击时进行验证。

创建 Validation 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件、一个 Button 控件、一个 RangeValidator 控件。如果验证失败，文本 "The value must be from 1 to 100!" 将会显示在 RangeValidator 控件中：

实例

```
<html>
<body>

<form runat="server">
<p>Enter a number from 1 to 100:
<asp:TextBox id="tbox1" runat="server" />
<br /><br />
<asp:Button Text="Submit" runat="server" />
</p>

<p>
<asp:RangeValidator
ControlToValidate="tbox1"
MinimumValue="1"
MaximumValue="100"
Type="Integer"
Text="The value must be from 1 to 100!"
runat="server" />
</p>
</form>

</body>
</html>
```

ASP.NET Web Forms - 事件

事件句柄是一种针对给定事件来执行代码的子例程。

ASP.NET - 事件句柄

请看下面的代码：

```
<%
lbl1.Text="The date and time is " & now()
%>

<html>
<body>
<form runat="server">
<h3><asp:label id="lbl1" runat="server" /></h3>
</form>
</body>
</html>
```

上面的代码将在何时被执行？答案是："不知道..."。

Page_Load 事件

Page_Load 事件是 ASP.NET 可理解的众多事件之一。Page_Load 事件会在页面加载时被触发， ASP.NET 将自动调用 Page_Load 子例程，并执行其中的代码：

实例

```
<script runat="server">
Sub Page_Load
lbl1.Text="The date and time is " & now()
End Sub
</script>

<html>
<body>
<form runat="server">
<h3><asp:label id="lbl1" runat="server" /></h3>
</form>
</body>
</html>
```

演示实例？

注释：Page_Load 事件不包含对象引用或事件参数！

Page.IsPostBack 属性

Page_Load 子例程会在页面每次加载时运行。如果您只想在页面第一次加载时执行 Page_Load 子例程中的代码，那么您可以使用 Page.IsPostBack 属性。如果 Page.IsPostBack 属性设置为 false，则页面第一次被载入，如果设置为 true，则页面被传回到服务器（比如，通过点击表单上的按钮）：

实例

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
lbl1.Text="The date and time is " & now()
end if
End Sub

Sub submit(s As Object, e As EventArgs)
lbl2.Text="Hello World!"
End Sub
</script>

<html>
<body>
<form runat="server">
<h3><asp:label id="lbl1" runat="server" /></h3>
<h3><asp:label id="lbl2" runat="server" /></h3>
<asp:button text="Submit" onclick="submit" runat="server" />
</form>
</body>
</html>
```

演示实例？

上面的实例仅在页面第一次加载时显示 "The date and time is...." 消息。当用户点击 Submit 按钮是，submit 子例程将会在第二个 label 中写入 "Hello World!"，但是第一个 label 中的日期和时间不会改变。

ASP.NET Web Forms - HTML 表单

所有的服务器控件都必须出现在 <form> 标签中，<form> 标签必须包含 runat="server" 属性。

ASP.NET Web 表单

所有的服务器控件都必须出现在 <form> 标签中，<form> 标签必须包含 runat="server" 属性。runat="server" 属性表明该表单必须在服务器上进行处理。同时也表明了包含在它内部的控件可被服务器脚本访问：

```
<form runat="server">
...HTML + server controls
</form>
```

注释：该表单总是被提交到自身页面。如果您指定了一个 action 属性，它会被忽略。如果您省略了 metion 属性，它将会默认设置 method="post"。同时，如果您没有指定 name 和 id 属性，它们会由 ASP.NET 自动分配。

注释：一个 .aspx 页面只能包含一个 <form runat="server"> 控件！

如果您在一个包含不带有 name、method、action 或 id 属性的表单的 .aspx 页面中选择查看源代码，您会看到 ASP.NET 添加这些属性到表单上了，如下所示：

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
...some code
</form>
```

提交表单

表单通常通过点击按钮来提交。ASP.NET 中的 Button 服务器控件的格式如下：

```
<asp:Button id="id" text="label" OnClick="sub" runat="server" />
```

id 属性为按钮定义了一个唯一的名称，text 属性为按钮分配了一个标签。onClick 事件句柄规定了一个要执行的已命名的子例程。

在下面的实例中，我们在 .aspx 文件中声明了一个 Button 控件。点击按钮运行改变按钮上文本的子例程：

实例

ASP.NET Web Forms - 维持 ViewState

通过在您的 Web Form 中维持对象的 ViewState（视图状态），您可以省去大量的编码工作。

维持 ViewState（视图状态）

在经典 ASP 中，当一个表单被提交时，所有的表单值都会被清空。假设您提交了一个带有大量信息的表单，而服务器返回了一个错误。您不得不回到表单改正信息。您点击返回按钮，然后发生了什么......所有表单值都被清空了，您不得不重新开始所有的一切！站点没有维持您的 ViewState。

在 ASP .NET 中，当一个表单被提交时，表单会连同表单值一起出现在浏览器窗口中。如何做到的呢？这是因为 ASP .NET 维持了您的 ViewState。 ViewState 会在页面被提交到服务器时表明它的状态。这个状态是通过在带有 <form runat="server"> 控件的每个页面上放置一个隐藏域定义的。源代码如下所示：

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
<input type="hidden" name="__VIEWSTATE"
value="dDwtNTI0ODU5MDE1Ozs+ZBCF2ryjMpeVgUrY2eTj79HNl4Q=" />

.....some code

</form>
```

维持 ViewState 是 ASP.NET Web Forms 的默认设置。如果您想不维持 ViewState，请在 .aspx 页面顶部包含指令 <%@ Page EnableViewState="false" %>，或者向任意控件添加属性 EnableViewState="false" 。

请看下面的 .aspx 文件。它演示了"老"的运行方式。当您点击提交按钮，表单值将会消失：

实例

```
<html>
<body>

<form action="demo_classicasp.aspx" method="post">
Your name: <input type="text" name="fname" size="20">
<input type="submit" value="Submit">
</form>
<%
dim fname
fname=Request.Form("fname")
If fname<>"" Then
Response.Write("Hello " & fname & "!")
End If
%>

</body>
</html>
```

演示实例？

下面是新的 ASP .NET 方式。当您点击提交按钮，表单值不会消失：

实例

点击实例的右边框架中的查看源代码，您将看到 ASP .NET 已经在表单中添加了一个隐藏域来维持 ViewState。

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Hello " & txt1.Text & "!"
End Sub
</script>

<html>
<body>

<form runat="server">
Your name: <asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

ASP.NET Web Forms - TextBox 控件

TextBox 控件用于创建用户可输入文本的文本框。

TextBox 控件

TextBox 控件用于创建用户可输入文本的文本框。

TextBox 控件的特性和属性列在我们的 WebForms 控件参考手册页面。

下面的实例演示了您可能会用到的 TextBox 控件的一些属性：

实例

```
<html>
<body>

<form runat="server">

A basic TextBox:
<asp:TextBox id="tb1" runat="server" />
<br /><br />

A password TextBox:
<asp:TextBox id="tb2" TextMode="password" runat="server" />
<br /><br />

A TextBox with text:
<asp:TextBox id="tb4" Text="Hello World!" runat="server" />
<br /><br />

A multiline TextBox:
<asp:TextBox id="tb3" TextMode="multiline" runat="server" />
<br /><br />

A TextBox with height:
<asp:TextBox id="tb6" rows="5" TextMode="multiline"
runat="server" />
<br /><br />

A TextBox with width:
<asp:TextBox id="tb5" columns="30" runat="server" />

</form>

</body>
</html>
```

演示实例？

添加脚本

当表单被提交时，TextBox 控件的内容和设置可能会被服务器脚本修改。表单可通过点击一个按钮或当用户修改 TextBox 控件的值的时候进行提交。

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件、一个 Button 控件和一个 Label 控件。当提交按钮被触发时，submit 子例程将被执行。submit 子例程将写入一行文本到 Label 控件中：

实例

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Your name is " & txt1.Text
End Sub
</script>

<html>
<body>

<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件和一个 Label 控件。当您修改了 TextBox 中的值，并且在 TextBox 外部点击（或者按下了 Tab 键）时，change 子例程将会被执行。change 子例程将写入一行文本到 Label 控件中：

实例

```
<script runat="server">
Sub change(sender As Object, e As EventArgs)
lbl1.Text="You changed text to " & txt1.Text
End Sub
</script>

<html>
<body>

<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server"
text="Hello World!"
ontextchanged="change" autopostback="true"/>
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

ASP.NET Web Forms - Button 控件

Button 控件用于显示一个下压按钮。

Button 控件

Button 控件用于显示一个下压按钮。下压按钮可能是一个提交按钮或者是一个命令按钮。在默认情况下，这个控件是提交按钮。

提交按钮没有命令名称，当它被点击时，它会把页面传回到服务器。您可以编写一些事件句柄，当提交按钮被点击时，用来控制动作的执行。

命令按钮有命令名称，并且允许您在页面上创建多个 Button 控件。您可以编写一些时间句柄，当命令按钮被点击时，用来控制动作的执行。

Button 控件的特性和属性列在我们的 WebForms 控件参考手册页面。

下面的实例演示了一个简单的 Button 控件：

```
<html>
<body>

<form runat="server">
<asp:Button id="b1" Text="Submit" runat="server" />
</form>

</body>
</html>
```

添加脚本

表单通常通过点击按钮进行提交。

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件、一个 Button 控件和一个 Label 控件。当提交按钮被触发时，submit 子例程将被执行。submit 子例程将写入一行文本到 Label 控件中：

实例

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Your name is " & txt1.Text
End Sub
</script>

<html>
<body>

<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

ASP.NET Web Forms - 数据绑定

我们可以使用数据绑定（Data Binding）来完成带可选项的列表，这些可选项来自某个导入的数据源，比如数据库、XML 文件或者脚本。

数据绑定

下面的控件是支持数据绑定的列表控件：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

以上每个控件的可选项通常是在一个或者多个 asp:ListItem 控件中定义，如下：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="countrylist" runat="server">
<asp:ListItem value="N" text="Norway" />
<asp:ListItem value="S" text="Sweden" />
<asp:ListItem value="F" text="France" />
<asp:ListItem value="I" text="Italy" />
</asp:RadioButtonList>
</form>

</body>
</html>
```

然而，我们可以使用某种独立的数据源进行数据绑定，比如数据库、XML 文件或者脚本，通过数据绑定来填充列表的可选项。

通过使用导入的数据源，数据从 HTML 中分离出来，并且对可选项的修改都是在独立的数据源中完成的。

在下面的三个章节中，我们将描述如何从脚本化的数据源中绑定数据。

ASP.NET Web Forms - ArrayList 对象

ArrayList 对象是包含单个数据值的项目的集合。

尝试一下 - 实例

ArrayList DropDownList

ArrayList RadioButtonList

创建 ArrayList

ArrayList 对象是包含单个数据值的项目的集合。

通过 Add() 方法向 ArrayList 添加项目。

下面的代码创建了一个名为 mycountries 的 ArrayList 对象，并添加了四个项目：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
end if
end sub
</script>
```

在默认情况下，一个 ArrayList 对象包含 16 个条目。可通过 TrimToSize() 方法把 ArrayList 调整为最终尺寸：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
end if
end sub
</script>
```

通过 Sort() 方法，ArrayList 也能够按照字母顺序或者数字顺序进行排序：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
mycountries.Sort()
end if
end sub
</script>
```

要实现反向排序，请在 Sort() 方法后应用 Reverse() 方法：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
mycountries.Sort()
mycountries.Reverse()
end if
end sub
</script>
```

绑定数据到 ArrayList

ArrayList 对象可为下列的控件自动生成文本和值：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" />
</form>

</body>
</html>
```

然后添加创建列表的脚本，并且绑定列表中的值到 RadioButtonList 控件：

实例

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
mycountries.Sort()
rb.DataSource=mycountries
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" />
</form>

</body>
</html>
```

演示实例？

RadioButtonList 控件的 DataSource 属性被设置为该 ArrayList，它定义了这个 RadioButtonList 控件的数据源。RadioButtonList 控件的 DataBind() 方法把 RadioButtonList 控件与数据源绑定在一起。

注释：数据值作为控件的 Text 和 Value 属性来使用。如需添加不同于 Text 的 Value，请使用 Hashtable 对象或者 SortedList 对象。

ASP.NET Web Forms - Hashtable 对象

Hashtable 对象包含用键/值对表示的项目。

尝试一下 - 实例

Hashtable RadiobuttonList 1

Hashtable RadiobuttonList 2

Hashtable DropDownList

创建 Hashtable

Hashtable 对象包含用键/值对表示的项目。键被用作索引，通过搜索键，可以实现对值的快速搜索。

通过 Add() 方法向 Hashtable 添加项目。

下面的代码创建了一个名为 mycountries 的 Hashtable 对象，并添加了四个元素：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New Hashtable
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
end if
end sub
</script>
```

数据绑定

Hashtable 对象可为下列的控件自动生成文本和值：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后添加创建列表的脚本，并且绑定列表中的值到 RadioButtonList 控件：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New Hashtable
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后我们添加一个子例程，当用户点击 RadioButtonList 控件中的某个项目时，该子例程会被执行。当某个单选按钮被点击时，label 中会出现一行文本：

实例

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New Hashtable
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub

sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

注释：您无法选择添加到 Hashtable 的项目的排序方式。如需对项目进行字母排序或者数字排序，请使用 SortedList 对象。

ASP.NET Web Forms - SortedList 对象

SortedList 对象结合了 ArrayList 对象和 Hashtable 对象的特性。

尝试一下 - 实例

SortedList RadiobuttonList 1

SortedList RadiobuttonList 2

SortedList DropDownList

SortedList 对象

SortedList 对象包含用键/值对表示的项目。SortedList 对象按照字母顺序或者数字顺序自动地对项目进行排序。

通过 Add() 方法向 SortedList 添加项目。通过 TrimToSize() 方法把 SortedList 调整为最终尺寸。

下面的代码创建了一个名为 mycountries 的 SortedList 对象，并添加了四个元素：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New SortedList
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
end if
end sub
</script>
```

数据绑定

SortedList 对象可为下列的控件自动生成文本和值：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后添加创建列表的脚本，并且绑定列表中的值到 RadioButtonList 控件：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New SortedList
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后我们添加一个子例程，当用户点击 RadioButtonList 控件中的某个项目时，该子例程会被执行。当某个单选按钮被点击时，label 中会出现一行文本：

实例

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New SortedList
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub

sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

ASP.NET Web Forms - XML 文件

我们可以绑定 XML 文件到列表控件。

一个 XML 文件

这里有一个名为 "countries.xml" 的 XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<countries>

<country>
<text>Norway</text>
<value>N</value>
</country>

<country>
<text>Sweden</text>
<value>S</value>
</country>

<country>
<text>France</text>
<value>F</value>
</country>

<country>
<text>Italy</text>
<value>I</value>
</country>

</countries>
```

查看这个 XML 文件：countries.xml

绑定 DataSet 到 List 控件

首先，导入 "System.Data" 命名空间。我们需要该命名空间与 DataSet 对象一起工作。把下面这条指令包含在 .aspx 页面的顶部：

```
<%@ Import Namespace="System.Data" %>
```

接着，为 XML 文件创建一个 DataSet，并在页面第一次加载时把这个 XML 文件载入 DataSet：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New DataSet
mycountries.ReadXml(MapPath("countries.xml"))
end if
end sub
```

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后添加创建 XML DataSet 的脚本，并且绑定 XML DataSet 中的值到 RadioButtonList 控件：

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New DataSet
mycountries.ReadXml(MapPath("countries.xml"))
rb.DataSource=mycountries
rb.DataValueField="value"
rb.DataTextField="text"
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
</form>

</body>
</html>
```

然后我们添加一个子例程，当用户点击 RadioButtonList 控件中的某个项目时，该子例程会被执行。当某个单选按钮被点击时，label 中会出现一行文本：

实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New DataSet
mycountries.ReadXml(MapPath("countries.xml"))
rb.DataSource=mycountries
rb.DataValueField="value"
rb.DataTextField="text"
rb.DataBind()
end if
end sub

sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例 ?

ASP.NET Web Forms - Repeater 控件

Repeater 控件用于显示被绑定在该控件上的项目的重复列表。

绑定 DataSet 到 Repeater 控件

Repeater 控件用于显示被绑定在该控件上的项目的重复列表。Repeater 控件可被绑定到数据库表、XML 文件或者其他项目列表。在这里，我们将演示如何绑定 XML 文件到 Repeater 控件。

在我们的实例中，我们将使用下面的 XML 文件（"cdcatalog.xml"）：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<catalog>
<cd>
<title>Empire Burlesque</title>
<artist>Bob Dylan</artist>
<country>USA</country>
<company>Columbia</company>
<price>10.90</price>
<year>1985</year>
</cd>
<cd>
<title>Hide your heart</title>
<artist>Bonnie Tyler</artist>
<country>UK</country>
<company>CBS Records</company>
<price>9.90</price>
<year>1988</year>
</cd>
<cd>
<title>Greatest Hits</title>
<artist>Dolly Parton</artist>
<country>USA</country>
<company>RCA</company>
<price>9.90</price>
<year>1982</year>
</cd>
<cd>
<title>Still got the blues</title>
<artist>Gary Moore</artist>
<country>UK</country>
<company>Virgin records</company>
<price>10.20</price>
<year>1990</year>
</cd>
<cd>
<title>Eros</title>
<artist>Eros Ramazzotti</artist>
<country>EU</country>
<company>BMG</company>
<price>9.90</price>
<year>1997</year>
</cd>
</catalog>
```

查看这个 XML 文件：cdcatalog.xml

首先，导入 "System.Data" 命名空间。我们需要该命名空间与 DataSet 对象一起工作。 把下面这条指令包含在 .aspx 页面的顶部：

```
<%@ Import Namespace="System.Data" %>
```

接着，为 XML 文件创建一个 DataSet，并在页面第一次加载时把这个 XML 文件载入 DataSet：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
end if
end sub
```

然后我们在 .aspx 页面中创建一个 Repeater 控件。<HeaderTemplate> 元素中的内容被首先呈现，并且在输出中仅出现一次，而 <ItemTemplate> 元素中的内容会对应 DataSet 中的每条 "record" 重复出现，最后，<FooterTemplate> 元素中的内容在输出中仅出现一次：

```
<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
...
</HeaderTemplate>

<ItemTemplate>
...
</ItemTemplate>

<FooterTemplate>
...
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

然后我们添加创建 DataSet 的脚本，并且绑定 mycdcatalog DataSet 到 Repeater 控件。然后使用 HTML 标签来填充 Repeater 控件，并通过 <%#Container.DataItem("fieldname")%> 绑定数据项目到 <ItemTemplate> 区域内的单元格中：

实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例？

使用 <AlternatingItemTemplate>

您可以在 <ItemTemplate> 元素后添加 <AlternatingItemTemplate> 元素，用来描述输出中交替行的外观。在下面的实例中，表格每隔一行就会显示为浅灰色的背景：

实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>

<AlternatingItemTemplate>
<tr bgcolor="#e8e8e8">
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</AlternatingItemTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例 ?

使用 <SeparatorTemplate>

<SeparatorTemplate> 元素用于描述每个记录之间的分隔符。在下面的实例中，每个表格行之间插入了一条水平线：

实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
<table border="0" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>

<SeparatorTemplate>
<tr>
<td colspan="6"><hr /></td>
</tr>
</SeparatorTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例？

ASP.NET Web Forms - DataList 控件

DataList 控件，类似于 Repeater 控件，用于显示绑定在该控件上的项目的重复列表。不过，DataList 控件会默认地在数据项目上添加表格。

绑定 DataSet 到 DataList 控件

DataList 控件，类似于 Repeater 控件，用于显示绑定在该控件上的项目的重复列表。不过，DataList 控件会默认地在数据项目上添加表格。DataList 控件可被绑定到数据库表、XML 文件或者其他项目列表。在这里，我们将演示如何绑定 XML 文件到 DataList 控件。

在我们的实例中，我们将使用下面的 XML 文件（"cdcatalog.xml"）：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<catalog>
<cd>
<title>Empire Burlesque</title>
<artist>Bob Dylan</artist>
<country>USA</country>
<company>Columbia</company>
<price>10.90</price>
<year>1985</year>
</cd>
<cd>
<title>Hide your heart</title>
<artist>Bonnie Tyler</artist>
<country>UK</country>
<company>CBS Records</company>
<price>9.90</price>
<year>1988</year>
</cd>
<cd>
<title>Greatest Hits</title>
<artist>Dolly Parton</artist>
<country>USA</country>
<company>RCA</company>
<price>9.90</price>
<year>1982</year>
</cd>
<cd>
<title>Still got the blues</title>
<artist>Gary Moore</artist>
<country>UK</country>
<company>Virgin records</company>
<price>10.20</price>
<year>1990</year>
</cd>
<cd>
<title>Eros</title>
<artist>Eros Ramazzotti</artist>
<country>EU</country>
<company>BMG</company>
<price>9.90</price>
<year>1997</year>
</cd>
</catalog>
```

查看这个 XML 文件：cdcatalog.xml

首先，导入 "System.Data" 命名空间。我们需要该命名空间与 DataSet 对象一起工作。 把下面这条指令包含在 .aspx 页面的顶部：

```
<%@ Import Namespace="System.Data" %>
```

接着，为 XML 文件创建一个 DataSet，并在页面第一次加载时把这个 XML 文件载入 DataSet：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
end if
end sub
```

然后我们在 .aspx 页面中创建一个 DataList 控件。<HeaderTemplate> 元素中的内容被首先呈现，并且在输出中仅出现一次，而 <ItemTemplate> 元素中的内容会对应 DataSet 中的每条 "record" 重复出现，最后，<FooterTemplate> 元素中的内容在输出中仅出现一次：

```
<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog" runat="server">

<HeaderTemplate>
...
</HeaderTemplate>

<ItemTemplate>
...
</ItemTemplate>

<FooterTemplate>
...
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

然后我们添加创建 DataSet 的脚本，并且绑定 mycdcatalog DataSet 到 DataList 控件。然后使用包含表头的 <HeaderTemplate>、包含要显示的数据项的 <ItemTemplate> 和包含文本的 <FooterTemplate> 来填充 DataList 控件。请注意，可设置 DataList 的 gridlines 属性为 "both" 来显示表格边框：

实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog"
gridlines="both" runat="server">

<HeaderTemplate>
My CD Catalog
</HeaderTemplate>

<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>

<FooterTemplate>
Copyright Hege Refsnes
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

演示实例？

使用样式

您也可以向 DataList 控件添加样式，让输出更加花哨：

实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog"
runat="server"
cellpadding="2"
cellspacing="2"
borderstyle="inset"
backcolor="#e8e8e8"
width="100%"
headerstyle-font-name="Verdana"
headerstyle-font-size="12pt"
headerstyle-horizontalalign="center"
headerstyle-font-bold="true"
itemstyle-backcolor="#778899"
itemstyle-forecolor="#ffffff"
footerstyle-font-size="9pt"
footerstyle-font-italic="true">

<HeaderTemplate>
My CD Catalog
</HeaderTemplate>

<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>

<FooterTemplate>
Copyright Hege Refsnes
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

演示实例？

使用 <AlternatingItemTemplate>

您可以在 <ItemTemplate> 元素后添加 <AlternatingItemTemplate> 元素，用来描述输出中交替行的外观。您可以在 DataList 控件内部对 <AlternatingItemTemplate> 区域的数据添加样式：

实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog"
runat="server"
cellpadding="2"
cellspacing="2"
borderstyle="inset"
backcolor="#e8e8e8"
width="100%"
headerstyle-font-name="Verdana"
headerstyle-font-size="12pt"
headerstyle-horizontalalign="center"
headerstyle-font-bold="True"
itemstyle-backcolor="#778899"
itemstyle-forecolor="#ffffff"
alternatingitemstyle-backcolor="#e8e8e8"
alternatingitemstyle-forecolor="#000000"
footerstyle-font-size="9pt"
footerstyle-font-italic="True">

<HeaderTemplate>
My CD Catalog
</HeaderTemplate>

<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>

<AlternatingItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</AlternatingItemTemplate>

<FooterTemplate>
&copy; Hege Refsnes
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

演示实例？

ASP.NET Web Forms - 数据库连接

ADO.NET 也是 .NET 框架的组成部分。ADO.NET 用于处理数据访问。通过 ADO.NET，您可以操作数据库。

尝试一下 - 实例

数据库连接 - 绑定到 DataList 控件

数据库连接 - 绑定到 Repeater 控件

什么是 ADO.NET？

- ADO.NET 是 .NET 框架的组成部分
- ADO.NET 由一系列用于处理数据访问的类组成
- ADO.NET 完全基于 XML
- ADO.NET 没有 Recordset 对象，这一点与 ADO 不同

创建数据库连接

在我们的实例中，我们将使用 Northwind 数据库。

首先，导入 "System.Data.OleDb" 命名空间。我们需要这个命名空间来操作 Microsoft Access 和其他 OLE DB 数据库提供商。我们将在 Page_Load 子例程中创建这个数据库的连接。我们创建一个 dbconn 变量，并为其赋值一个新的 OleDbConnection 类，这个类带有指示 OLE DB 提供商和数据库位置的连接字符串。然后我们打开数据库连接：

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
end sub
</script>
```

注释：这个连接字符串必须是没有折行的连续字符串！

创建数据库命令

为了指定需从数据库取回的记录，我们将创建一个 dbcomm 变量，并为其赋值一个新的 OleDbCommand 类。这个 OleDbCommand 类用于发出针对数据库表的 SQL 查询：

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
sql="SELECT * FROM customers"
dbcomm=New OleDbCommand(sql,dbconn)
end sub
</script>
```

创建 DataReader

OleDbDataReader 类用于从数据源中读取记录流。DataReader 是通过调用 OleDbCommand 对象的 ExecuteReader 方法来创建的：

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
sql="SELECT * FROM customers"
dbcomm=New OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader()
end sub
</script>
```

绑定到 Repeater 控件

然后，我们绑定 DataReader 到 Repeater 控件：

实例

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
sql="SELECT * FROM customers"
dbcomm=New OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader()
customers.DataSource=dbread
customers.DataBind()
dbread.Close()
dbconn.Close()
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="customers" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Companyname</th>
<th>Contactname</th>
<th>Address</th>
<th>City</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("companyname")%></td>
<td><%#Container.DataItem("contactname")%></td>
<td><%#Container.DataItem("address")%></td>
<td><%#Container.DataItem("city")%></td>
</tr>
</ItemTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例？

关闭数据库连接

如果不再需要访问数据库，请记得关闭 DataReader 和数据库连接：

```
dbread.Close()
dbconn.Close()
```

ASP.NET Web Forms - 母版页

母版页为您的网站的其他页面提供模版。

母版页

母版页允许您为您的 web 应用程序中的所有页面（或页面组）创建一致的外观和行为。

母版页为其他页面提供模版，带有共享的布局和功能。母版页为内容定义了可被内容页覆盖的占位符。输出结果是母版页和内容页的组合。

内容页包含您想要显示的内容。

当用户请求内容页时，ASP.NET 会对页面进行合并以生成结合了母版页布局和内容页内容的输出。

母版页实例

```
<%@ Master %>

<html>
<body>
<h1>Standard Header From Masterpage</h1>
<asp:ContentPlaceHolder id="CPH1" runat="server">
</asp:ContentPlaceHolder>
</body>
</html>
```

上面的母版页是一个为其他页面设计的普通 HTML 模版页。

@ Master 指令定义它为一个母版页。

母版页为单独的内容包含占位标签 **<asp:ContentPlaceHolder>**。

id="CPH1" 属性标识占位符，在相同母版页中允许多个占位符。

这个母版页被保存为 **"master1.master"**。

> 注释：母版页也能够包含代码，允许动态的内容。

内容页实例

```
<%@ Page MasterPageFile="master1.master" %>

<asp:Content ContentPlaceHolderId="CPH1" runat="server">
<h2>Individual Content</h2>
<p>Paragraph 1</p>
<p>Paragraph 2</p>
</asp:Content>
```

上面的内容页是站点中独立的内容页中的一个。

@ Page 指令定义它为一个标准的内容页。

内容页包含内容标签 **<asp:Content>**，该标签引用了母版页（ContentPlaceHolderId="CPH1"）。

这个内容页被保存为 **"mypage1.aspx"**。

当用户请求该页面时，ASP.NET 就会将母版页与内容页进行合并。

点击这里显示 mypage1.aspx

> 注释：内容文本必须位于 <asp:Content> 标签内部。标签外的内容文本是不允许的。

带控件的内容页

```
<%@ Page MasterPageFile="master1.master" %>

<asp:Content ContentPlaceHolderId="CPH1" runat="server">
<h2>W3CSchool</h2>
<form runat="server">
<asp:TextBox id="textbox1" runat="server" />
<asp:Button id="button1" runat="server" text="Button" />
</form>
</asp:Content>
```

上面的内容页演示了如何把 .NET 控件插入内容页，就像插入一个普通的页面中。

点击这里显示 mypage2.aspx

ASP.NET Web Forms - 导航

ASP.NET 带有内建的导航控件。

网站导航

维护大型网站的菜单是困难而且费时的。

在 ASP.NET 中，菜单可存储在文件中，这样易于维护。文件通常名为 **web.sitemap**，并且被存放在网站的根目录下。

此外，ASP.NET 有三个心的导航控件：

- Dynamic menus
- TreeViews
- Site Map Path

Sitemap 文件

在本教程中，使用下面的 sitemap 文件：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<siteMap>
<siteMapNode title="Home" url="/aspnet/w3home.aspx">
<siteMapNode title="Services" url="/aspnet/w3services.aspx">
<siteMapNode title="Training" url="/aspnet/w3training.aspx"/>
<siteMapNode title="Support" url="/aspnet/w3support.aspx"/>
</siteMapNode>
</siteMapNode>
</siteMap>
```

创建 sitemap 文件的规则：

- XML 文件必须包含 围绕内容的 <siteMap> 标签
- <siteMap> 标签只能有一个 <siteMapNode> 子节点（"home" 页面）
- 每个 <siteMapNode> 可以有多个子节点（网页）
- 每个 <siteMapNode> 带有定义页面标题和 URL 的属性

> 注释：sitemap 文件必须位于站点根目录下，URL 属性必须相对于该根目录。

动态菜单

<asp:Menu> 控件可显示标准的站点导航菜单。

代码实例：

```
<asp:SiteMapDataSource id="nav1" runat="server" />

<form runat="server">
<asp:Menu runat="server" DataSourceId="nav1" />
</form>
```

上面实例中的 **<asp:Menu>** 控件是一个供服务器创建导航菜单的占位符。

控件的数据源由 **DataSourceId** 属性定义。 **id="nav1"** 把数据源连接到 **<asp:SiteMapDataSource>** 控件。

<asp:SiteMapDataSource> 控件自动连接默认的 sitemap 文件（**web.sitemap**）。

TreeView

<asp:TreeView> 控件可显示多级导航菜单。

这种菜单看上去像一棵带有枝叶的树，可通过 + 或 - 符号来打开或关闭。

代码实例：

```
<asp:SiteMapDataSource id="nav1" runat="server" />

<form runat="server">
<asp:TreeView runat="server" DataSourceId="nav1" />
</form>
```

上面实例中的 **<asp:TreeView>** 控件是一个供服务器创建导航菜单的占位符。

控件的数据源由 **DataSourceId** 属性定义。 **id="nav1"** 把数据源连接到 **<asp:SiteMapDataSource>** 控件。

<asp:SiteMapDataSource> 控件自动连接默认的 sitemap 文件（**web.sitemap**）。

SiteMapPath

SiteMapPath 控件可显示指向当前页面的指针（导航路径）。该路径显示为指向上级页面的可点击链接。

与 TreeView 和 Menu 控件不同，SiteMapPath 控件不使用 SiteMapDataSource。SiteMapPath 控件默认使用 web.sitemap 文件。

> 提示：如果 SiteMapPath 没有正确显示，很可能是由于 web.sitemap 文件中存在 URL 错误（打印错误）。

代码实例：

```
<form runat="server">
<asp:SiteMapPath runat="server" />
</form>
```

上面实例中的 **<asp:SiteMapPath>** 控件是一个供服务器创建导航菜单的占位符。

Web Pages 参考手册

ASP.NET Web Pages - 类

ASP.NET 类参考手册

方法	描述
AsBool(), AsBool(true|false)	转换字符串值为布尔值（true/false）。如果字符串不能转换为true/false，则返回 false 或者其他规定的值。
AsDateTime(), AsDateTime(*value*)	转换字符串值为日期/时间。返回 DateTime。如果字符串不能转换为日期/时间，则返回 MinValue 或者其他规定的值。
AsDecimal(), AsDecimal(*value*)	转换字符串值为十进制值。如果字符串不能转换为十进制值，则返回 0.0 或者其他规定的值。
AsFloat(), AsFloat(*value*)	转换字符串值为浮点数。如果字符串不能转换为浮点数，则返回 0.0 或者其他规定的值。
AsInt(), AsInt(*value*)	转换字符串值为整数。如果字符串不能转换成整数，则返回 0 或者其他规定的值。
Href(*path [, param1 [, param2]]*)	从带有可选的附加路径部分的本地文件路径创建一个浏览器兼容的 URL。
Html.Raw(*value*)	Renders *value* 呈现为 HTML 标记，而不是呈现为 HTML 编码输出。
IsBool(), IsDateTime(), IsDecimal(), IsFloat(), IsInt()	如果该值可以从字符串转换为指定的类型，则返回 true。
IsEmpty()	如果对象或者变量没有值，则返回 true。
IsPost	如果请求是 POST，则返回 true。（初始请求通常是 GET。）
Layout	规定布局页面的路径应用于此页面。

PageData[*key*], PageData[*index*], Page	在当前请求的页面、布局页面、部分页面之间包含共数据。您可以使用动态来对相同的数据进行属性问。
RenderBody()	(Layout pages) 呈现没有布局页面任何命名区域的容页的内容Renders the content of a content pa that is not in any named sections.
RenderPage(*path*, *values*) RenderPage(*path[, param1 [, param2]]*)	呈现使用了规定的路径和选的额外数据的内容页。可以通过 position（实例或者 key（实例 2）从 PageData 获取额外参数值。
RenderSection(*sectionName [, required* = true|false*]*)	(Layout pages) 呈现一个名字的内容区域。设置 *required* 让一个区域为必非可选的。
Request.Cookies[*key*]	获取或者设置 HTTP co 的值。
Request.Files[*key*]	Gets 在当前请求中上传件。
Request.Form[*key*]	获取在表单中 post 的数（作为字符串）。Request.Form 和 Request.QueryString 者 [*key*] 检查。
Request.QueryString[*key*]	获取 URL 查询字符串中的数据。Request.Form Request.QueryString 者 [*key*] 检查。
Request.Unvalidated(*key*) Request.Unvalidated().QueryString|Form|Cookies|Headers[*key*]	有选择地禁用请求验证单元素、查询字符串值、cookie、header 值）。验证默认是开启的，防户提交标记或者其他潜在危险内容。
Response.AddHeader(*name*, *value*)	在应答中添加一个 HTT 务器响应头。
Response.OutputCache(*seconds [, sliding] [, varyByParams]*)	Caches 在指定时间的页出缓存。设置 *sliding* 来每个页面的访问超时时设置 *varyByParams* 为

	页面的每个不同的查询字 串缓存不同版本的页面。
Response.Redirect(*path*)	重定向浏览器请求到一个 的位置。
Response.SetStatus(*httpStatusCode*)	设置HTTP状态代码发送 览器。
Response.WriteBinary(*data [, mimetype]*)	写入 *data* 内容响应可选 MIME类型。
Response.WriteFile(*file*)	写入文件内容响应。
@section(*sectionName*) { *content* }	（布局页面）定义一个有 字的内容区域。
Server.HtmlDecode(*htmlText*)	解码一个HTML编码的字 串。
Server.HtmlEncode(*text*)	为呈现在 HTML 标记中 符串编码。
Server.MapPath(*virtualPath*)	为指定的虚拟路径返回服 器的物理路径。
Server.UrlDecode(*urlText*)	解码URL文本。
Server.UrlEncode(*text*)	URL文本编码。
Session[*key*]	获取或设置一个存在的值 直到用户关闭浏览器。
ToString()	显示一个用字符串表示的 象的值。
UrlData[*index*]	从 URL 获取额外的数据 如，*/MyPage/ExtraDat*

ASP.NET Web Pages - WebSecurity 对象

描述

WebSecurity 对象提供 ASP.NET Web Pages 应用程序的安全性和认证。

通过 WebSecurity 对象，您可以创建用户帐户，登录和注销用户，重置或者更改密码，以及其他更多与安全性相关的功能。

WebSecurity 对象参考手册 - 属性

属性	描述
CurrentUserId	获取当前登录用户的 ID。
CurrentUserName	获取当前登录用户的名称。
HasUserId	如果当前有用户 ID，则返回 true。
IsAuthenticated	如果当前用户是登录的，则返回 true。

WebSecurity 对象参考手册 - 方法

方法	描述
ChangePassword()	为指定的用户更改密码。
ConfirmAccount()	使用帐户确认令牌确认帐户。
CreateAccount()	创建一个新的用户帐户。
CreateUserAndAccount()	创建一个新的用户帐户。
GeneratePasswordResetToken()	生成一个密码重置令牌，可以在电子邮件中发送给用户以便用户可以重设密码。
GetCreateDate()	获取指定会员创建的时间。
GetPasswordChangeDate()	获取密码变更的日期和时间。
GetUserId()	根据用户名称获取用户 ID。
InitializeDatabaseConnection()	初始化 WebSecurity 系统（数据库）。
IsConfirmed()	检查用户是否已被确认。如果已确认，则返回 true。（例如，可通过电子邮件进行确认。）
IsCurrentUser()	检查当前用户的名称是否与指定用户名匹配。如果匹配，则返回 true。
Login()	设置身份验证令牌，登录用户。
Logout()	移除身份验证令牌，注销用户。
RequireAuthenticatedUser()	如果用户未通过身份验证，则设置 HTTP 状态为 401（未经授权）。
RequireRoles()	如果当前用户不是指定角色的成员，则设置 HTTP 状态为 401（未经授权）。
RequireUser()	如果当前用户不是指定用户名的用户，则设置 HTTP 状态为 401（未经授权）。
ResetPassword()	如果密码重置令牌是有效的，改变用户的密码为新密码。
UserExists()	检查指定的用户是否存在。

技术数据

名称	值
Class	WebMatrix.WebData.WebSecurity
Namespace	WebMatrix.WebData
Assembly	WebMatrix.WebData.dll

初始化 WebSecurity 数据库

如果您想在您的代码中使用 WebSecurity 对象，首先您必须创建或者初始化 WebSecurity 数据库。

在您的 Web 根目录下，创建一个名为 **_AppStart.cshtml** 的页面（如果已存在，则直接编辑页面）。

将下面的代码复制到文件中：

_AppStart.cshtml

```
@{
WebSecurity.InitializeDatabaseConnection("Users", "UserProfile", "UserId", "Email", true)
}
```

上面的代码将在每次网站（应用程序）启动时运行。它初始化了 WebSecurity 数据库。

"Users" 是 WebSecurity 数据库（Users.sdf）的名称。

"UserProfile" 是包含用户配置信息的数据库表的名称。

"UserId" 是包含用户 ID（主键）的列的名称。

"Email" 是包含用户名的列的名称。

最后一个参数 **true** 是一个布尔值，表示如果用户配置表和会员表不存在，则会自动创建表。如果不想自动创建表，应设置参数为 **false**。

> 虽然 **true** 表示自动创建数据库 表，但是数据库不会被自动创建。所以数据库必须存在。 |

WebSecurity 数据库

UserProfile 表为每个用户创建保存一条记录，用户 ID（主键）和用户名字（email）：

UserId	Email
1	john@johnson.net
2	peter@peterson.com
3	lars@larson.eut

Membership 表包含会员信息，比如用户是什么时候创建的，该会员是否已认证，会员是什么时候认证的，等等。

具体如下所示（一些列不显示）：

User Id	Create Date	Confirmation Token	Is Confirmed	Last Password Failure	Password	F
1	12.04.2012 16:12:17	NULL	True	NULL	AFNQhWfy....	1 1

注释：如果您想看到所有的列和内容，请打开数据库，看看里边的每个表。

简单的会员配置

在您使用 WebSecurity 对象时，如果您的站点没有配置使用 ASP.NET Web Pages 会员系统 **SimpleMembership**，可能会报错。

如果托管服务提供商的服务器的配置与您本地服务器的配置不同，也可能会报错。为了解决这个问题，请在网站的 Web.config 文件中添加以下元素：

ASP.NET Web Pages - Database 对象

ASP.NET Database 对象参考手册

方法	描述
Database.Execute(*SQLstatement [, parameters]*)	执行 SQL 语句 *SQLstatement*（带可选参数），比如 INSERT、DELETE 或者 UPDATE， 并且返回受影响的记录统计。
Database.GetLastInsertId()	返回最近插入行的标识列。
Database.Open(*filename*) Database.Open(*connectionStringName*)	使用 *Web.config* 文件中的连接字符串打开指定的数据库文件或者指定的数据库。
Database.OpenConnectionString(*connectionString*)	使用连接字符串打开一个数据库。（与 Database.Open 的差异是，Database.Open 使用的是连接字符串的名称，连接字符串的值在其他地方配置。）
Database.Query(*SQLstatement[, parameters]*)	使用 SQL 语句 *SQLstatement*（带可选参数）查询数据库，并返回结果集合。
Database.QuerySingle(*SQLstatement [, parameters]*)	执行 SQL 语句 *SQLstatement*（带可选参数），并返回单条记录。
Database.QueryValue(*SQLstatement [, parameters]*)	执行 SQL 语句 *SQLstatement*（带可选参数），并返回单个值。

ASP.NET Web Pages - WebMail 对象

通过 WebMail 对象，您可以很容易地从网页上发送电子邮件。

描述

WebMail 对象为 ASP.NET Web Pages 提供了使用 SMTP（Simple Mail Transfer Protocol 简单邮件传输协议）发送邮件的功能。

实例

请查看 WebPages Email 章节中的实例。

WebMail 对象参考手册 - 属性

属性	描述
SmtpServer	用于发送电子邮件的 SMTP 服务器的名称。
SmtpPort	服务器用来发送 SMTP 电子邮件的端口。
EnableSsl	如果服务器使用 SSL（Secure Socket Layer 安全套接层）加密，则值为 true。
UserName	用于发送电子邮件的 SMTP 电子邮件账户的名称。
Password	SMTP 电子邮件账户的密码。
From	在发件地址栏显示的电子邮件（通常与 UserName 相同）。

WebMail 对象参考手册 - 方法

方法	描述
Send()	向 SMTP 服务器发送需要传送的电子邮件信息。

Send() 方法有以下参数：

参数	类型	描述
to	String	收件人（用分号分隔）
subject	String	邮件主题
body	String	邮件正文

Send() 方法有以下可选参数：

参数	类型	描述
from	String	发件人
cc	String	需要抄送的电子邮件地址（用分号分隔）
filesToAttach	Collection	附件名
isBodyHtml	Boolean	如果邮件正文是 HTML 格式的，则为 true
additionalHeaders	Collection	附加的标题

技术数据

名称	值
Class	System.Web.Helpers.WebMail
Namespace	System.Web.Helpers
Assembly	System.Web.Helpers.dll

初始化 **WebMail** 帮助器

要使用 WebMail 帮助器，您必须能访问 SMTP 服务器。SMTP 是电子邮件的"输出"部分。如果您使用的是虚拟主机，您可能已经知道 SMTP 服务器的名称。如果您使用的是公司网络工作，您公司的 IT 部门会给您一个名称。如果您是在家工作，你也许可以使用普通的电子邮件服务提供商。

为了发送一封电子邮件，您将需要：

- SMTP 服务器的名称
- 端口号（通常是 25 ）
- 电子邮件的用户名
- 电子邮件的密码

在您的 Web 根目录下，创建一个名为 **_AppStart.cshtml** 的页面（如果已存在，则直接编辑页面）。

将下面的代码复制到文件中：

_AppStart.cshtml

```
@{
WebMail.SmtpServer = "smtp.example.com";
WebMail.SmtpPort = 25;
WebMail.EnableSsl = false;
WebMail.UserName = "support@example.com";
WebMail.Password = "password";
WebMail.From = "john@example.com"
}
```

上面的代码将在每次网站（应用程序）启动时运行。它对 **WebMail** 对象赋了初始值。

请替换：

将 **smtp.example.com** 替换成您要用来发送电子邮件的 SMTP 服务器的名称。

将 **25** 替换成服务器用来发送 SMTP 事务（电子邮件）的端口号。

如果服务器使用 SSL（Secure Socket Layer 安全套接层）加密，请将 **false** 替换成 true。

将 **support@example.com** 替换成用来发送电子邮件的 SMTP 电子邮件账户的名称。

将 **password** 替换成 SMTP 电子邮件账户的密码。

将 **john@example** 替换成显示在发件地址栏中的电子邮件。

> 在您的 AppStart 文件中，您不需要启动 **WebMail** 对象，但是在调用 **WebMail.Send()** 方法之前，您必须设置这些属性。

ASP.NET Web Pages - 更多帮助器

ASP.NET 帮助器 - 对象参考手册

Analytics 对象参考手册（Google）

Helper	描述
Analytics.GetGoogleHtml(*webPropertyId*)	为指定的 ID 呈现 Google Analytics JavaScript 代码。
Analytics.GetStatCounterHtml(*project*, *security*)	为指定的项目呈现 StatCounter Analytics JavaScript 代码。
Analytics.GetYahooHtml(*account*)	为指定的账号呈现 Yahoo Analytics JavaScript 代码。

Bing 对象参考手册

Helper	描述
Bing.SearchBox(*[boxWidth]*)	给 Bing 传递搜索。您可以设置 Bing.SiteUrl 和 Bing.SiteTitle 属性来设定站点搜索和搜索框的标题，通常是在 *_AppStart* 页面设置这些属性。
Bing.AdvancedSearchBox(*[, boxWidth] [, resultWidth] [, resultHeight] [, themeColor] [, locale]*)	用可选的格式显示 Bing 搜索结果在页面上。您可以设置 Bing.SiteUrl 和 Bing.SiteTitle 属性来设定站点搜索和搜索框的标题，通常是在 *_AppStart* 页面设置这些属性。

Chart 对象参考手册

Helper	描述
Chart(*width*, *height [, template] [, templatePath]*)	初始化图表。
Chart.AddLegend(*[title] [, name]*)	给图表添加一个图例。
Chart.AddSeries(*[name] [, chartType] [, chartArea] [, axisLabel] [, legend] [, markerStep] [, xValue] [, xField] [, yValues] [, yFields] [, options]*)	给图表添加一系列数据。

Crypto 对象参考手册

Helper	描述
Crypto.Hash(*string [, algorithm]*) Crypto.Hash(*bytes [, algorithm]*)	返回指定数据的哈希。默认算法是 sha256。

Facebook 对象参考手册

Helper	描述
Facebook.LikeButton(*href [, buttonLayout] [, showFaces] [, width] [, height] [, action] [, font] [, colorScheme] [, refLabel]*)	让 Facebook 用户连接到网页。

FileUpload 对象参考手册

Helper	描述
FileUpload.GetHtml(*[initialNumberOfFiles] [, allowMoreFilesToBeAdded] [, includeFormTag] [, addText] [, uploadText]*)	为上传文件呈现 UI。

GamerCard 对象参考手册

Helper	描述
GamerCard.GetHtml(*gamerTag*)	呈现指定的 Xbox gamer 标签。

Gravatar 对象参考手册

Helper	描述
Gravatar.GetHtml(*email [, imageSize] [, defaultImage] [, rating] [, imageExtension] [, attributes]*)	为指定的电子邮件地址呈现 Gravatar 图像。

Json 对象参考手册

Helper	描述
Json.Encode(*object*)	用 JavaScript Object Notation (JSON) 把数据对象转换为字符串。
Json.Decode(*string*)	转换 JSON 编码的输入字符串为您指定的数据对象。

LinkShare 对象参考手册

Helper	描述
LinkShare.GetHtml(*pageTitle [, pageLinkBack] [, twitterUserName] [, additionalTweetText] [, linkSites]*)	使用指定的标题和可选的 URL 呈现社会网络链接。

ModelState 对象参考手册

Helper	描述
ModelStateDictionary.AddError(*key*, *errorMessage*)	关联错误信息和一个表单域。使用 ModelState 帮助器访问成员。
ModelStateDictionary.AddFormError(*errorMessage*)	关联错误信息和一个表单。使用 ModelState 帮助器访问成员。
ModelStateDictionary.IsValid	如果没有验证错误，返回 true。使用 ModelState 帮助器访问成员。

ObjectInfo 对象参考手册

Helper	描述
ObjectInfo.Print(*value [, depth] [, enumerationLength]*)	呈现一个对象和所有子对象的属性和值。

Recaptcha 对象参考手册

Helper	描述
Recaptcha.GetHtml(*[, publicKey] [, theme] [, language] [, tabIndex]*)	呈现 reCAPTCHA 验证测试。
ReCaptcha.PublicKey ReCaptcha.PrivateKey	设置 reCAPTCHA 服务的公共和私有密钥。通常是在 *_AppStart* 页面设置这些属性。
ReCaptcha.Validate(*[, privateKey]*)	返回 reCAPTCHA 测试结果。
ServerInfo.GetHtml()	Renders 呈现有关 ASP.NET Web Pages 的状态信息。

Twitter 对象参考手册

Helper	描述
Twitter.Profile(*twitterUserName*)	为指定的用户呈现 Twitter 流。
Twitter.Search(*searchQuery*)	为指定的搜索文本呈现 Twitter 流。

Video 对象参考手册

Helper	描述
Video.Flash(*filename [, width, height]*)	为指定的文件呈现宽度和高度可选的 Flash 视频播放。
Video.MediaPlayer(*filename [, width, height]*)	为指定的文件呈现宽度和高度可选 的 Windows Media 播放器。
Video.Silverlight(*filename*, *width*, *height*)	为指定的 *.xap* 文件呈现所需的宽度和高度 的 Silverlight 播放器。

WebCache 对象参考手册

Helper	描述
WebCache.Get(*key*)	通过 *key* 返回指定的对象，如果对象未找到则返回 null。
WebCache.Remove(*key*)	通过 *key* 从缓存中删除指定的对象。
WebCache.Set(*key*, *value [, minutesToCache] [, slidingExpiration]*)	通过 *key* 把 *value* 放置到指定名称的缓存中。

WebGrid 对象参考手册

Helper	描述
WebGrid(*data*)	Creates a 使用查询数据创建一个新的 WebGrid 对象。
WebGrid.GetHtml()	Renders markup 显示数据在 HTML 表格中。
WebGrid.Pager()	为 WebGrid 对象呈现一个页面。

WebImage 对象参考手册

Helper	描述
WebImage(*path*)	从指定的路径加载一个图像。
WebImage.AddImagesWatermark(*image*)	为指定图像加水印。
WebImage.AddTextWatermark(*text*)	为图像添加指定文本。
WebImage.FlipHorizontal() WebImage.FlipVertical()	水平/垂直翻转图像
WebImage.GetImageFromRequest()	当图像被传送到一个文件上传页面时，加载图像。
WebImage.Resize(*width*, *height*)	调整图像大小。
WebImage.RotateLeft() WebImage.RotateRight()	向左或向右旋转图像。
WebImage.Save(*path [, imageFormat]*)	保存图像到指定路径。

ASP.NET MVC - 参考手册

类

类	描述
AcceptVerbsAttribute	表示一个特性，该特性指定操作方法将响应的 HTTP 谓词。
ActionDescriptor	提供有关操作方法的信息，比如操作方法的名称、控制器、参数、特性和筛选器。
ActionExecutedContext	提供 ActionFilterAttribute 类的 ActionExecuted 方法的上下文。
ActionExecutingContext	提供 ActionFilterAttribute 类的 ActionExecuting 方法的上下文。
ActionFilterAttribute	表示筛选器特性的基类。
ActionMethodSelectorAttribute	表示一个用于影响操作方法选择的特性。
ActionNameAttribute	表示一个用于操作的名称的特性。
ActionNameSelectorAttribute	表示一个可影响操作方法选择的特性。
ActionResult	封装一个操作方法的结果并用于代表该操作方法执行框架级操作。
AdditionalMetadataAttribute	提供一个类，该类实现 IMetadataAware 接口以支持其他元数据。
AjaxHelper	表示支持在视图中呈现 AJAX 方案中的 HTML。
AjaxHelper(TModel)	表示支持在强类型视图中呈现 AJAX 方案中的 HTML。
AjaxRequestExtensions	表示一个类，该类对 HttpRequestBase 类进行了扩展，在其中添加了确定 HTTP 请求是否为 AJAX 请求的功能。
AllowHtmlAttribute	通过跳过属性的请求验证，允许请求在模型绑定过程中包含 HTML 标记。（强烈建议应用程序显式检查所有禁用请求验证的模型，以防止脚本攻击。）
AreaRegistration	提供在一个 ASP.NET MVC 应用程序内注册一个或多个区域的方式。
AreaRegistrationContext	对在 ASP.NET MVC 应用程序内注册某个区域时所需的信息进行封装。
AssociatedMetadataProvider	提供用于实现元数据提供程序的抽象类。

AssociatedValidatorProvider	为用于实现验证提供程序的类提供抽象类。
AsyncController	为异步控制器提供基类。
AsyncTimeoutAttribute	表示一个特性，该特性用于设置异步方法的超时值（以毫秒为单位）。
AuthorizationContext	对使用 AuthorizeAttribute 特性时所需的信息进行封装。
AuthorizeAttribute	表示一个特性，该特性用于限制调用方对操作方法的访问。
BindAttribute	表示一个特性，该特性用于提供有关应如何进行模型绑定到参数的详细信息。
BuildManagerCompiledView	表示在视图引擎呈现视图之前由 BuildManager 类编译的视图的基类。
BuildManagerViewEngine	为视图引擎提供基类。
ByteArrayModelBinder	映射浏览器请求到字节数组。
ChildActionOnlyAttribute	表示一个特性，该特性用于指示操作方法只应作为子操作进行调用。
ChildActionValueProvider	表示子操作中的值的值提供程序。
ChildActionValueProviderFactory	表示用于为子操作创建值提供程序对象的工厂。
ClientDataTypeModelValidatorProvider	返回客户端数据类型模型验证程序。
CompareAttribute	提供用于比较某个模型的两个属性的特性。
ContentResult	表示用户定义的内容类型，该类型是操作方法的结果。
Controller	提供用于响应对 ASP.NET MVC 网站所进行的 HTTP 请求的方法。
ControllerActionInvoker	表示一个类，该类负责调用控制器的操作方法。
ControllerBase	表示所有 MVC 控制器的基类。
ControllerBuilder	表示一个类，该类负责动态生成控制器。
ControllerContext	封装有关与指定的 RouteBase 和 ControllerBase 实例匹配的 HTTP 请求的信息。
ControllerDescriptor	封装描述控制器的信息，比如控制器的名称、类型和操作。
ControllerInstanceFilterProvider	将控制器添加到 FilterProviderCollection 实例。

CustomModelBinderAttribute	表示一个调用自定义模型联编程序的特性。
DataAnnotationsModelMetadata	为数据模型的公共元数据、DataAnnotationsModelMetadataProvider 类和 DataAnnotationsModelValidator 类提供容器。
DataAnnotationsModelMetadataProvider	实现 ASP.NET MVC 的默认模型元数据提供程序。
DataAnnotationsModelValidator	提供模型验证程序。
DataAnnotationsModelValidator(TAttribute)	为指定的验证类型提供模型验证程序。
DataAnnotationsModelValidatorProvider	实现 ASP.NET MVC 的默认验证提供程序。
DataErrorInfoModelValidatorProvider	为错误信息模型验证程序提供容器。
DefaultControllerFactory	表示默认情况下已注册的控制器工厂。
DefaultModelBinder	映射浏览器请求到数据对象。该类提供模型联编程序的具体实现。
DefaultViewLocationCache	表示视图位置的内存缓存。
DependencyResolver	为实现 IDependencyResolver 或公共服务定位器 IServiceLocator 接口的依赖关系解析程序提供一个注册点。
DependencyResolverExtensions	提供 GetService 和 GetServices 的类型安全实现。
DictionaryValueProvider(TValue)	表示值提供程序的基类，这些值提供程序的值来自实现 IDictionary(TKey, TValue) 接口的集合。
EmptyModelMetadataProvider	为不需要元数据的数据模型提供空的元数据提供程序。
EmptyModelValidatorProvider	为不需要验证程序的模型提供空的验证提供程序。
EmptyResult	表示一个不执行任何操作的结果，比如一个不返回任何内容的控制器操作方法。
ExceptionContext	P提供使用 HandleErrorAttribute 类的上下文。
ExpressionHelper	提供用于从表达式中获取模型名称的帮助器类。
FieldValidationMetadata	为客户端字段验证元数据提供容器。
FileContentResult	将二进制文件的内容发送到响应。
FilePathResult	将文件的内容发送到响应。

FileResult	表示一个用于将二进制文件内容发送到响应的基类。
FileStreamResult	使用 Stream 实例将二进制内容发送到响应。
Filter	表示一个元数据类，它包含对一个或多个筛选器接口的实现、筛选器顺序和筛选器范围的引用。
FilterAttribute	表示操作和结果筛选器特性的基类。
FilterAttributeFilterProvider	定义筛选器特性的筛选器提供程序。
FilterInfo	封装有关可用的操作筛选器的信息。
FilterProviderCollection	表示应用程序的筛选器提供程序的集合。
FilterProviders	为筛选器提供一个注册点。
FormCollection	包含应用程序的表单值提供程序。
FormContext	对验证和处理 HTML 表单中的输入数据所需的信息进行封装。
FormValueProvider	表示 NameValueCollection 对象中包含的表单值的值提供程序。
FormValueProviderFactory	表示一个类，该类负责创建表单值提供程序对象的新实例。
GlobalFilterCollection	表示一个包含所有全局筛选器的类。
GlobalFilters	表示全局筛选器集合。
HandleErrorAttribute	表示一个特性，该特性用于处理由操作方法引发的异常。
HandleErrorInfo	封装有关处理由操作方法引发的错误的信息。
HiddenInputAttribute	表示一个特性，该特性用于指示是否应将属性值或字段值呈现为隐藏的 input 元素。
HtmlHelper	表示支持在视图中呈现 HTML 控件。
HtmlHelper(TModel)	表示支持在强类型视图中呈现 HTML 控件。
HttpDeleteAttribute	表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP DELETE 请求。
HttpFileCollectionValueProvider	表示要用于来自 HTTP 文件集合的值的值提供程序。
HttpFileCollectionValueProviderFactory	表示一个类，该类负责创建 HTTP 文件集合值提供程序对象的新实例。

HttpGetAttribute	表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP GET 请求。
HttpNotFoundResult	定义一个用于指示未找到所请求资源的对象。
HttpPostAttribute	表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP POST 请求。
HttpPostedFileBaseModelBinder	将模型绑定到已发布的文件。
HttpPutAttribute	表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP PUT 请求。
HttpRequestExtensions	扩展 HttpRequestBase 类，该类包含客户端在 Web 请求中发送的 HTTP 值。
HttpStatusCodeResult	提供一种用于返回带特定 HTTP 响应状态代码和说明的操作结果的方法。
HttpUnauthorizedResult	表示未经授权的 HTTP 请求的结果。
JavaScriptResult	将 JavaScript 内容发送到响应。
JsonResult	表示一个类，该类用于将 JSON 格式的内容发送到响应。
JsonValueProviderFactory	启用操作方法以发送和接收 JSON 格式的文本，并将 JSON 文本以模型绑定方式传递给操作方法的参数。
LinqBinaryModelBinder	映射浏览器请求到 LINQ Binary 对象。
ModelBinderAttribute	表示一个特性，该特性用于将模型类型关联到模型-生成器类型。
ModelBinderDictionary	表示一个类，该类包含应用程序的所有模型联编程序（按联编程序类型列出）。
ModelBinderProviderCollection	为模型联编程序提供程序提供一个容器。
ModelBinderProviders	为模型联编程序提供程序提供一个容器。
ModelBinders	提供对应用程序的模型联编程序的全局访问。
ModelBindingContext	提供运行模型联编程序的上下文。
ModelClientValidationEqualToRule	为发送到浏览器的相等验证规则提供一个容器。
ModelClientValidationRangeRule	为发送到浏览器的范围验证规则提供一个容器。
ModelClientValidationRegexRule	为发送到浏览器的正则表达式客户端验证规则提供一个容器。
ModelClientValidationRemoteRule	为发送到浏览器的远程验证规则提供一个容器。

ModelClientValidationRequiredRule	为必填字段的客户端验证提供一个容器。
ModelClientValidationRule	为发送到浏览器的客户端验证规则提供一个基类容器。
ModelClientValidationStringLengthRule	为发送到浏览器的字符串长度验证规则提供一个容器。
ModelError	表示在模型绑定期间发生的错误。
ModelErrorCollection	ModelError 实例的集合。
ModelMetadata	为数据模型的公共元数据、ModelMetadataProvider 类和 ModelValidator 类提供容器。
ModelMetadataProvider	为自定义元数据提供程序提供抽象基类。
ModelMetadataProviders	为当前的 ModelMetadataProvider 实例提供容器。
ModelState	将模型绑定的状态封装到操作方法参数的一个属性或操作方法参数本身。
ModelStateDictionary	表示将已发送表单绑定到操作方法（其中包括验证信息）的尝试的状态。
ModelValidationResult	为验证结果提供容器。
ModelValidator	提供用于实现验证逻辑的基类。
ModelValidatorProvider	为模型提供验证程序的列表。
ModelValidatorProviderCollection	为验证提供程序的列表提供一个容器。
ModelValidatorProviders	为当前验证提供程序提供容器。
MultiSelectList	表示一个项列表，用户可从该列表中选择多个项。
MvcFilter	在派生类中实现时，提供一个元数据类，它包含对一个或多个筛选器接口的实现、筛选器顺序和筛选器范围的引用。
MvcHandler	选择将处理 HTTP 请求的控制器。
MvcHtmlString	表示不应再次进行编码的 HTML 编码的字符串。
MvcHttpHandler	验证并处理 HTTP 请求。
MvcRouteHandler	创建一个实现 IHttpHandler 接口的对象并向该对象传递请求上下文。
MvcWebRazorHostFactory	创建 MvcWebPageRazorHost 文件的实例。
NameValueCollectionExtensions	扩展 NameValueCollection 对象，以便能够将集合复制到指定字典。

NameValueCollectionValueProvider	表示值提供程序的基类，这些值提供程序的值来自 NameValueCollection 对象。
NoAsyncTimeoutAttribute	为 AsyncTimeoutAttribute 特性提供便利包装。
NonActionAttribute	表示一个特性，该特性用于指示控制器方法不是操作方法。
OutputCacheAttribute	表示一个特性，该特性用于标记将缓存其输出的操作方法。
ParameterBindingInfo	封装与将操作方法参数绑定到数据模型相关的信息。
ParameterDescriptor	包含描述参数的信息。
PartialViewResult	表示一个用于将部分视图发送到响应的基类。
PreApplicationStartCode	为 ASP.NET Razor 应用程序预启动代码提供注册点。
QueryStringValueProvider	表示 NameValueCollection 对象中包含的查询字符串的值提供程序。
QueryStringValueProviderFactory	表示一个类，该类负责创建查询字符串值提供程序对象的新实例。
RangeAttributeAdapter	提供 RangeAttribute 特性的适配器。
RazorView	表示用于创建具有 Razor 语法的视图的类。
RazorViewEngine	表示一个用于呈现使用 ASP.NET Razor 语法的 Web 页面的视图引擎。
RedirectResult	通过重定向到指定的 URI 来控制对应用程序操作的处理。
RedirectToRouteResult	表示使用指定的路由值字典来执行重定向的结果。
ReflectedActionDescriptor	包含描述反射的操作方法的信息。
ReflectedControllerDescriptor	包含描述反射的控制器的信息。
ReflectedParameterDescriptor	包含描述反射的操作方法参数的信息。
RegularExpressionAttributeAdapter	提供 RegularExpressionAttribute 特性的适配器。
RemoteAttribute	提供使用 jQuery 验证插件远程验证程序的特性。
RequiredAttributeAdapter	提供 RequiredAttributeAttribute 特性的适配器。
RequireHttpsAttribute	表示一个特性，该特性用于强制通过 HTTPS 重新发送不安全的 HTTP 请求。

ResultExecutedContext	提供 ActionFilterAttribute 类的 OnResultExecuted 方法的上下文。
ResultExecutingContext	提供 ActionFilterAttribute 类的 OnResultExecuting 方法的上下文。
RouteCollectionExtensions	扩展 RouteCollection 对象以进行 MVC 路由。
RouteDataValueProvider	表示实现 IDictionary(TKey, TValue) 接口的对象中包含的路由数据的值提供程序。
RouteDataValueProviderFactory	表示用来创建路由数据值提供程序对象的工厂。
SelectList	表示一个列表，用户可从该列表中选择一个项。
SelectListItem	表示 SelectList 类的实例中的选定项。
SessionStateAttribute	指定控制器的会话状态。
SessionStateTempDataProvider	为当前 TempDataDictionary 对象提供会话状态数据。
StringLengthAttributeAdapter	提供 StringLengthAttribute 特性的适配器。
TempDataDictionary	表示仅从一个请求保持到下一个请求的数据集。
TemplateInfo	封装有关当前模板上下文的信息。
UrlHelper	包含用于为应用程序内的 ASP.NET MVC 生成 URL 的方法。
UrlParameter	表示路由过程中 MvcHandler 类使用的可选参数。
ValidatableObjectAdapter	提供可验证的对象适配器。
ValidateAntiForgeryTokenAttribute	表示用于阻止伪造请求的特性。
ValidateInputAttribute	表示一个特性，该特性用于标记必须验证其输入的操作方法。
ValueProviderCollection	表示应用程序的值提供程序对象的集合。
ValueProviderDictionary	已过时。表示应用程序的值提供程序的字典。
ValueProviderFactories	表示值提供程序工厂对象的容器。
ValueProviderFactory	表示用来创建值提供程序对象的工厂。
ValueProviderFactoryCollection	表示应用程序的值提供程序工厂的集合。
ValueProviderResult	表示将一个值（如表单发送的值或查询字符串中的值）绑定到操作方法参数属性或

	绑定到该参数本身的结果。
ViewContext	封装与呈现视图相关的信息。
ViewDataDictionary	表示一个容器，该容器用于在控制器和视图之间传递数据。
ViewDataDictionary(TModel)	表示一个容器，该容器用于在控制器和视图之间传递强类型数据。
ViewDataInfo	对开发模板所使用的当前模板内容和与模板交互的 HTML 帮助器的相关信息进行封装。
ViewEngineCollection	表示对应用程序可用的视图引擎的集合。
ViewEngineResult	表示定位视图引擎的结果。
ViewEngines	表示对应用程序可用的视图引擎的集合。
ViewMasterPage	表示生成母版视图页所需的信息。
ViewMasterPage(TModel)	表示生成强类型母版视图页所需的信息。
ViewPage	表示将视图呈现为 Web Forms 页所需的属性和方法。
ViewPage(TModel)	表示将强类型视图呈现为 Web Forms 页所需的信息。
ViewResult	表示一个类，该类用于使用由 IViewEngine 对象返回的 IView 实例来呈现视图。
ViewResultBase	表示一个用于为视图提供模型并向响应呈现视图的基类。
ViewStartPage	提供可用于实现视图启动（母版）页的抽象类。
ViewTemplateUserControl	提供 TemplateInfo 对象的容器。
ViewTemplateUserControl(TModel)	提供 TemplateInfo 对象的容器。
ViewType	表示视图的类型。
ViewUserControl	表示生成用户控件所需的信息。
ViewUserControl(TModel)	表示生成强类型用户控件所需的信息。
VirtualPathProviderViewEngine	表示 IViewEngine 接口的抽象基类实现。
WebFormView	表示在 ASP.NET MVC 中生成 Web Forms 页时所需的信息。
WebFormViewEngine	表示一个用于向响应呈现 Web Forms 页的视图引擎。
WebViewPage	表示呈现使用 ASP.NET Razor 语法的视图所需的属性和方法。

WebViewPage(TModel)	表示呈现使用 ASP.NET Razor 语法的视图所需的属性和方法。

接口

接口	描述
IActionFilter	定义操作筛选器中使用的方法。
IActionInvoker	定义操作调用程序的协定，该调用程序用于调用一个操作以响应 HTTP 请求。
IAuthorizationFilter	定义授权筛选器所需的方法。
IClientValidatable	为 ASP.NET MVC 验证框架提供一种用于在运行时发现验证程序是否支持客户端验证的方法。
IController	定义控制器所需的方法。
IControllerActivator	对使用依赖关系注入来实例化控制器的方式进行精细控制。
IControllerFactory	定义控制器工厂所需的方法。
IDependencyResolver	定义可简化服务位置和依赖关系解析的方法。
IExceptionFilter	定义异常筛选器所需的方法。
IFilterProvider	提供用于查找筛选器的接口。
IMetadataAware	提供用于向 AssociatedMetadataProvider 类公开特性的接口。
IModelBinder	定义模型联编程序所需的方法。
IModelBinderProvider	定义用于为实现 IModelBinder 接口的类动态实现模型绑定的方法。
IMvcFilter	定义用于指定筛选器顺序以及是否允许多个筛选器的成员。
IResultFilter	定义结果筛选器所需的方法。
IRouteWithArea	将路由与 ASP.NET MVC 应用程序中的区域关联。
ITempDataProvider	定义临时数据提供程序的协定，这些临时数据提供程序用于存储要在下一个请求中查看的数据。
IUnvalidatedValueProvider	表示一个可跳过请求验证的 IValueProvider 接口。
IValueProvider	定义 ASP.NET MVC 中的值提供程序所需的方法。
IView	定义视图所需的方法。
IViewDataContainer	定义视图数据字典所需的方法。
IViewEngine	定义视图引擎所需的方法。
IViewLocationCache	定义在内存中缓存视图位置所需的方法。
IViewPageActivator	对使用依赖关系注入创建视图页的方式进行精细控制。

Web Forms 参考手册

ASP.NET Web Forms - HTML 服务器控件

HTML 服务器控件是服务器可理解的 HTML 标签。

HTML 服务器控件

ASP.NET 文件中的 HTML 元素，默认是作为文本进行处理的。要想让这些元素可编程，需向 HTML 元素中添加 runat="server" 属性。这个属性表示，该元素将被作为服务器控件进行处理。

注释：所有 HTML 服务器控件必须位于带有 runat="server" 属性的 <form> 标签内！

注释：ASP.NET 要求所有 HTML 元素必须正确关闭和正确嵌套。

HTML 服务器控件	描述
HtmlAnchor	控制 `<a>` HTML 元素
HtmlButton	控制 `<button>` HTML 元素
HtmlForm	控制 `<form>` HTML 元素
HtmlGeneric	控制其他未被具体的 HTML 服务器控件规定的 HTML 元素，比如 `<body>` 、 `<div>` 、 `<span>` 等。
HtmlImage	控制 `<image>` HTML 元素
HtmlInputButton	控制 `<input type="button">` 、 `<input type="submit">` 和 `<input type="reset">` HTML 元素
HtmlInputCheckBox	控制 `<input type="checkbox">` HTML 元素
HtmlInputFile	控制 `<input type="file">` HTML 元素
HtmlInputHidden	控制 `<input type="hidden">` HTML 元素
HtmlInputImage	控制 `<input type="image">` HTML 元素
HtmlInputRadioButton	控制 `<input type="radio">` HTML 元素
HtmlInputText	控制 `<input type="text">` 和 `<input type="password">` HTML 元素
HtmlSelect	控制 `<select>` HTML 元素
HtmlTable	控制 `<table>` HTML 元素
HtmlTableCell	控制 `<td>` 和 `<th>` HTML 元素
HtmlTableRow	控制 `<tr>` HTML 元素
HtmlTextArea	控制 `<textarea>` HTML 元素

ASP.NET Web Forms - Web 服务器控件

Web 服务器控件是服务器可理解的特殊 ASP.NET 标签。

Web 服务器控件

就像 HTML 服务器控件，Web 服务器控件也是在服务器上创建的，它们同样需要 runat="server" 属性才能生效。然而，Web 服务器控件没有必要映射任何已存在的 HTML 元素，它们可以表示更复杂的元素。

创建 Web 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

Web 服务器控件	描述
AdRotator	显示一个图形序列
Button	显示下压按钮
Calendar	显示日历
CalendarDay	calendar 控件中的一天
CheckBox	显示复选框
CheckBoxList	创建多选的复选框组
DataGrid	显示 grid 中数据源的字段
DataList	通过使用模版显示数据源中的项目
DropDownList	创建下拉列表
HyperLink	创建超链接
Image	显示图像
ImageButton	显示可点击的图像
Label	显示可编程的静态内容（使您对其内容应用样式）
LinkButton	创建超链接按钮
ListBox	创建单选或多选的下拉列表
ListItem	创建列表中的一个项目
Literal	显示可编程的静态内容（无法使您对其内容应用样式）
Panel	为其他控件提供容器
PlaceHolder	为由代码添加的控件预留空间
RadioButton	创建单选按钮
RadioButtonList	创建单选按钮组
BulletedList	创建项目符号格式的列表
Repeater	显示绑定到控件的项目的重复列表
Style	设置控件的样式
Table	创建表格
TableCell	创建表格单元格
TableRow	创建表格行
TextBox	创建文本框
Xml	显示 XML 文件或 XSL 转换的结果

ASP.NET Web Forms - Validation 服务器控件

Validation 服务器控件是用来验证用户输入的。

Validation 服务器控件

Validation 服务器控件用于验证输入控件的数据。如果数据未通过验证，则向用户显示错误消息。

创建 Validation 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

Validation 服务器控件	描述
CompareValidator	把一个输入控件的值与另一个输入控件的值或一个固定的值进行对比
CustomValidator	允许您编写一个方法，来处理输入值的验证
RangeValidator	检查用户输入值是否介于两个值之间
RegularExpressionValidator	确保输入控件的值匹配指定的模式
RequiredFieldValidator	使输入控件成为必需（必填）的字段
ValidationSummary	显示网页中所有验证错误的报告

W3School C语言教程

来源：C语言教程

整理：飞龙

C语言教程

C 简介

C 语言是一种通用的高级语言，最初是由丹尼斯·里奇在贝尔实验室为开发 UNIX 操作系统而设计的。C 语言最开始是于 1972 年在 DEC PDP-11 计算机上被首次实现。

在 1978 年，布莱恩·柯林汉（Brian Kernighan）和丹尼斯·里奇（Dennis Ritchie）制作了 C 的第一个公开可用的描述，现在被称为 K&R 标准。

UNIX 操作系统，C编译器，和几乎所有的 UNIX 应用程序都是用 C 语言编写的。由于各种原因，C 语言现在已经成为一种广泛使用的专业语言。

- 易于学习。
- 结构化语言。
- 它产生高效率的程序。
- 它可以处理底层的活动。
- 它可以在多种计算机平台上编译。

关于 C

- C 语言是为了编写 UNIX 操作系统而被发明的。
- C 语言是以 B 语言为基础的，B 语言大概是在 1970 年被引进的。
- C 语言标准是于 1988 年由美国国家标准协会（ANSI，全称 American National Standard Institute）制定的。
- 截至 1973 年，UNIX 操作系统完全使用 C 语言编写。
- 目前，C 语言是最广泛使用的系统程序设计语言。
- 大多数先进的软件都是使用 C 语言实现的。
- 当今最流行的 Linux 操作系统和 RBDMS MySQL 都是使用 C 语言编写的。

为什么要使用 C？

C 语言最初是用于系统开发工作，特别是组成操作系统的程序。由于 C 语言所产生的代码运行速度与汇编语言编写的代码运行速度几乎一样，所以采用 C 语言作为系统开发语言。下面列举几个使用 C 的实例：

- 操作系统
- 语言编译器
- 汇编器
- 文本编辑器
- 打印假脱机

- 网络驱动器
- 现代程序
- 数据库
- 语言解释器
- 实体工具

C 程序

一个 C 语言程序，可以是 3 行，也可以是数百万行，它可以写在一个或多个扩展名为 **".c"** 的文本文件中，例如，*hello.c*。您可以使用 **"vi"**、**"vim"** 或任何其他文本编辑器来编写您的 C 语言程序。

本教程假定您已经知道如何编辑一个文本文件，以及如何在程序文件中编写源代码。

C 环境设置

本地环境设置

如果您想要设置 C 语言环境，您需要确保电脑上有以下两款可用的软件，文本编辑器和 C 编译器。

文本编辑器

这将用于输入您的程序。文本编辑器包括 Windows Notepad、OS Edit command、Brief、Epsilon、EMACS 和 vim/vi。

文本编辑器的名称和版本在不同的操作系统上可能会有所不同。例如，Notepad 通常用于 Windows 操作系统上，vim/vi 可用于 Windows 和 Linux/UNIX 操作系统上。

通过编辑器创建的文件通常称为源文件，源文件包含程序源代码。C 程序的源文件通常使用扩展名 "**.c**"。

在开始编程之前，请确保您有一个文本编辑器，且有足够的经验来编写一个计算机程序，然后把它保存在一个文件中，编译并执行它。

C 编译器

写在源文件中的源代码是人类可读的源。它需要"编译"，转为机器语言，这样 CPU 可以按给定指令执行程序。

C 语言编译器用于把源代码编译成最终的可执行程序。这里假设您已经对编程语言编译器有基本的了解了。

最常用的免费可用的编译器是 GNU 的 C/C++ 编译器，如果您使用的是 HP 或 Solaris，则可以使用各自操作系统上的编译器。

以下部分将指导您如何在不同的操作系统上安装 GNU 的 C/C++ 编译器。这里同时提到 C/C++，主要是因为 GNU 的 gcc 编译器适合于 C 和 C++ 编程语言。

UNIX/Linux 上的安装

如果您使用的是 **Linux** 或 **UNIX**，请在命令行使用下面的命令来检查您的系统上是否安装了 GCC：

```
$ gcc -v
```

如果您的计算机上已经安装了 GNU 编译器，则会显示如下消息：

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

如果未安装 GCC，那么您必须按照 http://gcc.gnu.org/install/ 上的详细说明安装 GCC。

本教程是基于 Linux 编写的，所有给定的实例都已在 Cent OS Linux 系统上编译过。

Mac OS 上的安装

如果您使用的是 Mac OS X，最快捷的获取 GCC 的方法是从苹果的网站上下载 Xcode 开发环境，并按照安装说明进行安装。一旦安装上 Xcode，您就能使用 GNU 编译器。

Xcode 目前可从 developer.apple.com/technologies/tools/ 上下载。

Windows 上的安装

为了在 Windows 上安装 GCC，您需要安装 MinGW。为了安装 MinGW，请访问 MinGW 的主页 www.mingw.org，进入 MinGW 下载页面，下载最新版本的 MinGW 安装程序，命名格式为 MinGW-<version>.exe。

当安装 MinWG 时，您至少要安装 gcc-core、gcc-g++、binutils 和 MinGW runtime，但是一般情况下都会安装更多其他的项。

添加您安装的 MinGW 的 bin 子目录到您的 **PATH** 环境变量中，这样您就可以在命令行中通过简单的名称来指定这些工具。

当完成安装时，您可以从 Windows 命令行上运行 gcc、g++、ar、ranlib、dlltool 和其他一些 GNU 工具。

C 程序结构

在我们学习 C 语言的基本构建块之前，让我们先来看看一个最小的 C 程序结构，在接下来的章节中可以以此作为参考。

C Hello World 实例

C 程序主要包括以下部分：

- 预处理器指令
- 函数
- 变量
- 语句 & 表达式
- 注释

让我们看一段简单的代码，可以输出单词 "Hello World"：

```
#include <stdio.h>

int main()
{
   /* 我的第一个 C 程序 */
   printf("Hello, World! \n");

   return 0;
}
```

接下来我们讲解一下上面这段程序：

1. 程序的第一行 *#include <stdio.h>* 是预处理器指令，告诉 C 编译器在实际编译之前要包含 stdio.h 文件。
2. 下一行 *int main()* 是主函数，程序从这里开始执行。
3. 下一行 */.../* 将会被编译器忽略，这里放置程序的注释内容。它们被称为程序的注释。
4. 下一行 *printf(...)* 是 C 中另一个可用的函数，会在屏幕上显示消息 "Hello, World!"。
5. 下一行 **return 0;** 终止 main() 函数，并返回值 0。

编译 & 执行 C 程序

接下来让我们看看如何把源代码保存在一个文件中，以及如何编译并运行它。下面是简单的步骤：

1. 打开一个文本编辑器，添加上述代码。
2. 保存文件为 *hello.c*。

3. 打开命令提示符，进入到保存文件所在的目录。
4. 键入 *gcc hello.c*，输入回车，编译代码。
5. 如果代码中没有错误，命令提示符会跳到下一行，并生成 *a.out* 可执行文件。
6. 现在，键入 *a.out* 来执行程序。
7. 您可以看到屏幕上显示 *"Hello World"*。

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

请确保您的路径中已包含 gcc 编译器，并确保在包含源文件 hello.c 的目录中运行它。

C 基本语法

我们已经看过 C 程序的基本结构，这将有助于我们理解 C 语言的其他基本的构建块。

C 的令牌（Tokens）

C 程序由各种令牌组成，令牌可以是关键字、标识符、常量、字符串值，或者是一个符号。例如，下面的 C 语句包括五个令牌：

```
printf("Hello, World! \n");
```

这五个令牌分别是：

```
printf
(
"Hello, World! \n"
)
;
```

分号；

在 C 程序中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。

例如，下面是两个不同的语句：

```
printf("Hello, World! \n");
return 0;
```

注释

注释就像是 C 程序中的帮助文本，它们会被编译器忽略。它们以 / 开始，以字符 / 终止，如下所示：

```
/* 我的第一个 C 程序 */
```

您不能在注释内嵌套注释，注释也不能出现在字符串或字符值中。

标识符

C 标识符是用来标识变量、函数，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 _ 开始，后跟零个或多个字母、下划线和数字（0-9）。

C 标识符内不允许出现标点字符，比如 @、$ 和 %。C 是区分大小写的编程语言。因此，在 C 中，*Manpower* 和 *manpower* 是两个不同的标识符。下面列出几个有效的标识符：

```
mohd        zara    abc   move_name  a_123
myname50    _temp   j     a23b9      retVal
```

关键字

下表列出了 C 中的保留字。这些保留字不能作为常量名、变量名或其他标识符名称。

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

C 中的空格

只包含空格的行，可能带有注释，被称为空白行，C 编译器会完全忽略它。

在 C 中，空格用于描述空白符、制表符、换行符和注释。空格分隔语句的各个部分，让编译器能识别语句中的某个元素（比如 int）在哪里结束，下一个元素在哪里开始。因此，在下面的语句中：

```
int age;
```

在这里，int 和 age 之间必须至少有一个空格字符（通常是一个空白符），这样编译器才能够区分它们。另一方面，在下面的语句中：

```
fruit = apples + oranges;   // 获取水果的总数
```

fruit 和 =，或者 = 和 apples 之间没有必要有空格字符，但是为了增强可读性，您可以根据需要适当增加一些空格。

C 数据类型

在 C 语言中，数据类型指的是用于声明不同类型的变量或函数的一个广泛的系统。变量的类型决定了变量存储占用的空间，以及如何解释存储的位模式。

C 中的类型可分为以下几种：

类型	描述
基本类型：	它们是算术类型，包括两种类型：整数类型和浮点类型。
枚举类型：	它们也是算术类型，被用来定义在程序中只能赋予其一定的离散整数值的变量。
void 类型：	类型说明符 *void* 表明没有可用的值。
派生类型：	它们包括：指针类型、数组类型、结构类型、共用体类型和函数类型。

数组类型和结构类型统称为聚合类型。函数的类型指的是函数返回值的类型。在本章节接下来的部分我们将介绍基本类型，其他几种类型会在后边几个章节中进行讲解。

整数类型

下表列出了关于标准整数类型的存储大小和值范围的细节：

类型	存储大小	值范围
char	1 byte	-128 到 127 或 0 到 255
unsigned char	1 byte	0 到 255
signed char	1 byte	-128 到 127
int	2 或 4 bytes	-32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647
unsigned int	2 或 4 bytes	0 到 65,535 或 0 到 4,294,967,295
short	2 bytes	-32,768 到 32,767
unsigned short	2 bytes	0 到 65,535
long	4 bytes	-2,147,483,648 到 2,147,483,647
unsigned long	4 bytes	0 到 4,294,967,295

为了得到某个类型或某个变量在特定平台上的准确大小，您可以使用 **sizeof** 运算符。表达式 *sizeof(type)* 得到对象或类型的存储字节大小。下面的实例演示了获取 int 类型的大小：

```
#include <stdio.h>
#include <limits.h>

int main()
{
   printf("Storage size for int : %d \n", sizeof(int));

   return 0;
}
```

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

```
Storage size for int : 4
```

浮点类型

下表列出了关于标准浮点类型的存储大小、值范围和精度的细节：

类型	存储大小	值范围	精度
float	4 byte	1.2E-38 到 3.4E+38	6 位小数
double	8 byte	2.3E-308 到 1.7E+308	15 位小数
long double	10 byte	3.4E-4932 到 1.1E+4932	19 位小数

头文件 float.h 定义了宏，在程序中可以使用这些值和其他有关实数二进制表示的细节。下面的实例将输出浮点类型占用的存储空间以及它的范围值：

```
#include <stdio.h>
#include <float.h>

int main()
{
   printf("Storage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value: %d\n", FLT_DIG );

   return 0;
}
```

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

void 类型

void 类型指定没有可用的值。它通常用于以下三种情况下：

类型	描述
函数返回为空	C 中有各种函数都不返回值，或者您可以说它们返回空。不返回值的函数的返回类型为空。例如 **void exit (int status);**
函数参数为空	C 中有各种函数不接受任何参数。不带参数的函数可以接受一个 void。例如 **int rand(void);**
指针指向 **void**	类型为 void 的指针代表对象的地址，而不是类型。例如，内存分配函数 ***void* malloc(size_t size);** 返回指向 void 的指针，可以转换为任何数据类型。

如果现在您还是无法完全理解 void 类型，不用太担心，在后续的章节中我们将会详细讲解这些概念。

C 变量

变量其实只不过是程序可操作的存储区的名称。C 中每个变量都有指定的类型，类型决定了变量存储的大小和布局，值的范围可以存储在内存中，运算符可应用于变量上。

变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。大写字母和小写字母是不同的，因为 C 是大小写敏感的。基于前一章讲解的基本类型，将有以下集中基本变量类型：

类型	描述
char	通常是一个八位字节（一个字节）。这是一个整数类型。
int	对机器而言，整数的最自然的大小。
float	单精度浮点值。
double	双精度浮点值。
void	表示类型的缺失。

C 语言也允许定义各种其他类型的变量，比如枚举、指针、数组、结构、共用体等等，这将会在后续的章节中进行讲解，本章节我们先讲解基本变量类型。

C 中的变量定义

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

```
type variable_list;
```

在这里，**type** 必须是一个有效的 C 数据类型，可以是 char、w_char、int、float、double、bool 或任何用户自定义的对象，**variable_list** 可以由一个或多个标识符名称组成，多个标识符之间用逗号分隔。下面列出几个有效的声明：

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

行 **int i, j, k;** 声明并定义了变量 i、j 和 k，这指示编译器创建类型为 int 的名为 i、j、k 的变量。

变量可以在声明的时候被初始化（指定一个初始值）在他们的宣言。初始化器由一个等号，后跟一个常量表达式组成，如下所示：

```
type variable_name = value;
```

下面列举几个实例：

```
extern int d = 3, f = 5;    // d 和 f 的声明
int d = 3, f = 5;           // 定义并初始化 d 和 f
byte z = 22;                // 定义并初始化 z
char x = 'x';               // 变量 x 的值为 'x'
```

不带初始化的定义：带有静态存储持续时间的变量会被隐式初始化为 NULL（所有字节的值都是 0），其他所有变量的初始值是未定义的。

C 中的变量声明

变量声明向编译器保证变量以给定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

当您使用多个文件且只在其中一个文件中定义变量时（定义变量的文件在程序连接时是可用的），变量声明就显得非常有用。您可以使用 **extern** 关键字在任何地方声明一个变量。虽然您可以在程序中多次声明一个变量，但变量只能在某个文件、函数或代码块中被定义一次。

实例

尝试下面的实例，其中，变量在头部就已经被声明，但它们是在主函数内被定义和初始化的：

```
#include <stdio.h>

// 变量声明
extern int a, b;
extern int c;
extern float f;

int main ()
{
  /* 变量定义 */
  int a, b;
  int c;
  float f;

  /* 实际初始化 */
  a = 10;
  b = 20;

  c = a + b;
  printf("value of c : %d \n", c);

  f = 70.0/3.0;
  printf("value of f : %f \n", f);

  return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of c : 30
value of f : 23.333334
```

同样的，在函数声明时，提供一个函数名，而函数的实际定义则可以在任何地方进行。例如：

```
// 函数声明
int func();

int main()
{
    // 函数调用
    int i = func();
}

// 函数定义
int func()
{
    return 0;
}
```

C 中的左值（Lvalues）和右值（Rvalues）

C 中有两种类型的表达式：

1. 左值（**lvalue**）：指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边。
2. 右值（**rvalue**）：术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能

对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边。下面是一个有效的语句：

```
int g = 20;
```

但是下面这个就不是一个有效的语句，会生成编译时错误：

```
10 = 20;
```

C 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量。

常量可以是任何的基本数据类型，比如整数常量、浮点常量、字符常量，或字符串字面值，也有枚举常量。

常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

```
212         /* 合法的 */
215u        /* 合法的 */
0xFeeL      /* 合法的 */
078         /* 非法的：8 不是八进制的数字 */
032UU       /* 非法的：不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85         /* 十进制 */
0213       /* 八进制 */
0x4b       /* 十六进制 */
30         /* 整数 */
30u        /* 无符号整数 */
30l        /* 长整数 */
30ul       /* 无符号长整数 */
```

浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含小数点、指数，或同时包含两者。当使用指数形式表示时，必须包含整数部分、小数部分，或同时包含两者。带符号的指数是用 e 或 E 引入的。

下面列举几个浮点常量的实例：

```
3.14159       /* 合法的 */
314159E-5L    /* 合法的 */
510E          /* 非法的：不完整的指数 */
210f          /* 非法的：没有小数或指数 */
.e55          /* 非法的：缺少整数或分数 */
```

字符常量

字符常量是括在单引号中，例如，'x' 可以存储在 **char** 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）。

在 C 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：

转义序列	含义
\	\ 字符
\'	' 字符
\"	" 字符
\?	? 字符
\a	警报铃声
\b	退格键
\f	换页符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ooo	一到三位的八进制数
\xhh . . .	一个或多个数字的十六进制数

下面的实例显示了一些转义序列字符：

```
#include <stdio.h>

int main()
{
   printf("Hello\tWorld\n\n");

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello   World
```

字符串常量

字符串字面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

定义常量

在 C 中，有两种简单的定义常量的方式：

1. 使用 **#define** 预处理器。
2. 使用 **const** 关键字。

#define 预处理器

下面是使用 #define 预处理器定义常量的形式：

```
#define identifier value
```

具体请看下面的实例：

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main()
{

   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of area : 50
```

const 关键字

您可以使用 **const** 前缀声明指定类型的常量，如下所示：

```
const type variable = value;
```

具体请看下面的实例：

```
#include <stdio.h>

int main()
{
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of area : 50
```

请注意，把常量定义为大写字母形式，是一个很好的编程实践。

C 存储类

存储类定义 C 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C 程序中可用的存储类：

- auto
- register
- static
- extern

auto 存储类

auto 存储类是所有局部变量默认的存储类。

```
{
   int mount;
   auto int month;
}
```

上面的实例定义了两个带有相同存储类的变量，auto 只能用在函数内，即 auto 只能修饰局部变量。

register 存储类

register 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

```
{
   register int  miles;
}
```

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

static 存储类

static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。

static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

在 C 编程中，当 **static** 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

```
#include <stdio.h>

/* 函数声明 */
void func(void);

static int count = 5; /* 全局变量 */

main()
{
   while(count--)
   {
      func();
   }
   return 0;
}
/* 函数定义 */
void func( void )
{
   static int i = 5; /* 局部静态变量 */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

可能您现在还无法理解这个实例，因为我已经使用了函数和全局变量，这两个概念目前为止还没进行讲解。即使您现在不能完全理解，也没有关系，后续的章节我们会详细讲解。当上面的代码被编译和执行时，它会产生下列结果：

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

extern 存储类

extern 存储类用于提供一个全局变量的引用，它对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 *extern* 来得到已定义的变量或函数的引用。可以这么理解，*extern* 是用来在另一个文件中声明一个全局变量或函数。

extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

第一个文件：**main.c**

```
#include <stdio.h>

int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

第二个文件：**support.c**

```
#include <stdio.h>

extern int count;

void write_extern(void)
{
   printf("count is %d\n", count);
}
```

在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.c 中定义的 *count*。现在，编译这两个文件，如下所示：

```
 $gcc main.c support.c
```

这会产生 **a.out** 可执行程序，当程序被执行时，它会产生下列结果：

```
5
```

C 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C 语言内置了丰富的运算符，并提供了以下类型的运算符：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

本教程将逐一介绍算术运算符、关系运算符、逻辑运算符、按位运算符、赋值运算符和其他运算符。

算术运算符

下表显示了 C 语言支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

实例

请看下面的实例，了解 C 语言中所有可用的算术运算符：

```
#include <stdio.h>

main()
{
   int a = 21;
   int b = 10;
   int c ;

   c = a + b;
   printf("Line 1 - c 的值是 %d\n", c );
   c = a - b;
   printf("Line 2 - c 的值是 %d\n", c );
   c = a * b;
   printf("Line 3 - c 的值是 %d\n", c );
   c = a / b;
   printf("Line 4 - c 的值是 %d\n", c );
   c = a % b;
   printf("Line 5 - c 的值是 %d\n", c );
   c = a++;
   printf("Line 6 - c 的值是 %d\n", c );
   c = a--;
   printf("Line 7 - c 的值是 %d\n", c );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 21
Line 7 - c 的值是 22
```

关系运算符

下表显示了 C 语言支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

实例

请看下面的实例，了解 C 语言中所有可用的关系运算符：

```
#include <stdio.h>

main()
{
   int a = 21;
   int b = 10;
   int c ;

   if( a == b )
   {
      printf("Line 1 - a 等于 b\n" );
   }
   else
   {
      printf("Line 1 - a 不等于 b\n" );
   }
   if ( a < b )
   {
      printf("Line 2 - a 小于 b\n" );
   }
   else
   {
      printf("Line 2 - a 不小于 b\n" );
   }
   if ( a > b )
   {
      printf("Line 3 - a 大于 b\n" );
   }
   else
   {
      printf("Line 3 - a 不大于 b\n" );
   }
   /* 改变 a 和 b 的值 */
   a = 5;
   b = 20;
   if ( a <= b )
   {
      printf("Line 4 - a 小于或等于 b\n" );
   }
   if ( b >= a )
   {
      printf("Line 5 - b 大于或等于 b\n" );
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - a 不等于 b
Line 2 - a 不小于 b
Line 3 - a 大于 b
Line 4 - a 小于或等于 b
Line 5 - b 大于或等于 b
```

逻辑运算符

下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 **A** 的值为 1，变量 **B** 的值为 0，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A || B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

实例

请看下面的实例，了解 C 语言中所有可用的逻辑运算符：

```
#include <stdio.h>

main()
{
   int a = 5;
   int b = 20;
   int c ;

   if ( a && b )
   {
      printf("Line 1 - 条件为真\n" );
   }
   if ( a || b )
   {
      printf("Line 2 - 条件为真\n" );
   }
   /* 改变 a 和 b 的值 */
   a = 0;
   b = 10;
   if ( a && b )
   {
      printf("Line 3 - 条件为真\n" );
   }
   else
   {
      printf("Line 3 - 条件不为真\n" );
   }
   if ( !(a && b) )
   {
      printf("Line 4 - 条件为真\n" );
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

位运算符

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

下表显示了 C 语言支持的位运算符。假设变量 **A** 的值为 60，变量 **B** 的值为 13，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果。	(~A) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

请看下面的实例，了解 C 语言中所有可用的位运算符：

```
#include <stdio.h>

main()
{

   unsigned int a = 60;    /* 60 = 0011 1100 */
   unsigned int b = 13;    /* 13 = 0000 1101 */
   int c = 0;

   c = a & b;       /* 12 = 0000 1100 */
   printf("Line 1 - c 的值是 %d\n", c );

   c = a | b;       /* 61 = 0011 1101 */
   printf("Line 2 - c 的值是 %d\n", c );

   c = a ^ b;       /* 49 = 0011 0001 */
   printf("Line 3 - c 的值是 %d\n", c );

   c = ~a;          /*-61 = 1100 0011 */
   printf("Line 4 - c 的值是 %d\n", c );

   c = a << 2;     /* 240 = 1111 0000 */
   printf("Line 5 - c 的值是 %d\n", c );

   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Line 6 - c 的值是 %d\n", c );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

赋值运算符

下表列出了 C 语言支持的赋值运算符：

<table>
<tr><th>运算符</th><th>描述</th><th>实例</th></tr>
<tr><td>=</td><td>简单的赋值运算符，把右边操作数的值赋给左边操作数</td><td>C = A + B 将把 A + B 的值赋给 C</td></tr>
<tr><td>+=</td><td>加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数</td><td>C += A 相当于 C = C + A</td></tr>
<tr><td>-=</td><td>减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数</td><td>C -= A 相当于 C = C - A</td></tr>
<tr><td>*=</td><td>乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数</td><td>C = A 相当于 C = C A</td></tr>
<tr><td>/=</td><td>除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数</td><td>C /= A 相当于 C = C / A</td></tr>
<tr><td>%=</td><td>求模且赋值运算符，求两个操作数的模赋值给左边操作数</td><td>C %= A 相当于 C = C % A</td></tr>
<tr><td><<=</td><td>左移且赋值运算符</td><td>C <<= 2 等同于 C = C << 2</td></tr>
<tr><td>>>=</td><td>右移且赋值运算符</td><td>C >>= 2 等同于 C = C >> 2</td></tr>
<tr><td>&=</td><td>按位与且赋值运算符</td><td>C &= 2 等同于 C = C & 2</td></tr>
<tr><td>^=</td><td>按位异或且赋值运算符</td><td>C ^= 2 等同于 C = C ^ 2</td></tr>
<tr><td>|=</td><td>按位或且赋值运算符</td><td>C |= 2 等同于 C = C | 2</td></tr>
</table>

实例

请看下面的实例，了解 C 语言中所有可用的赋值运算符：

```
#include <stdio.h>

main()
{
   int a = 21;
   int c ;

   c =  a;
   printf("Line 1 - =  运算符实例，c 的值 = %d\n", c );

   c +=  a;
   printf("Line 2 - += 运算符实例，c 的值 = %d\n", c );

   c -=  a;
   printf("Line 3 - -= 运算符实例，c 的值 = %d\n", c );

   c *=  a;
   printf("Line 4 - *= 运算符实例，c 的值 = %d\n", c );

   c /=  a;
   printf("Line 5 - /= 运算符实例，c 的值 = %d\n", c );

   c  = 200;
   c %=  a;
   printf("Line 6 - %= 运算符实例，c 的值 = %d\n", c );

   c <<=  2;
   printf("Line 7 - <<= 运算符实例，c 的值 = %d\n", c );

   c >>=  2;
   printf("Line 8 - >>= 运算符实例，c 的值 = %d\n", c );

   c &=  2;
   printf("Line 9 - &= 运算符实例，c 的值 = %d\n", c );

   c ^=  2;
   printf("Line 10 - ^= 运算符实例，c 的值 = %d\n", c );

   c |=  2;
   printf("Line 11 - |= 运算符实例，c 的值 = %d\n", c );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - =  运算符实例，c 的值 = 21
Line 2 - += 运算符实例，c 的值 = 42
Line 3 - -= 运算符实例，c 的值 = 21
Line 4 - *= 运算符实例，c 的值 = 441
Line 5 - /= 运算符实例，c 的值 = 21
Line 6 - %= 运算符实例，c 的值 = 11
Line 7 - <<= 运算符实例，c 的值 = 44
Line 8 - >>= 运算符实例，c 的值 = 11
Line 9 - &= 运算符实例，c 的值 = 2
Line 10 - ^= 运算符实例，c 的值 = 0
Line 11 - |= 运算符实例，c 的值 = 2
```

杂项运算符 ? sizeof & 三元

下表列出了 C 语言支持的其他一些重要的运算符，包括 **sizeof** 和 **? :**。

运算符	描述	实例
sizeof()	返回变量的大小。	sizeof(a) 将返回 4，其中 a 是整数。
&	返回变量的地址。	&a; 将给出变量的实际地址。
*	指向一个变量。	*a; 将指向一个变量。
? :	条件表达式	如果条件为真 ? 则值为 X : 否则值为 Y

实例

请看下面的实例，了解 C 语言中所有可用的杂项运算符：

```
#include <stdio.h>

main()
{
   int a = 4;
   short b;
   double c;
   int* ptr;

   /* sizeof 运算符实例 */
   printf("Line 1 - 变量 a 的大小 = %d\n", sizeof(a) );
   printf("Line 2 - 变量 b 的大小 = %d\n", sizeof(b) );
   printf("Line 3 - 变量 c 的大小 = %d\n", sizeof(c) );

   /* & 和 * 运算符实例 */
   ptr = &a;    /* 'ptr' 现在包含 'a' 的地址 */
   printf("a 的值是 %d\n", a);
   printf("*ptr 是 %d\n", *ptr);

   /* 三元运算符实例 */
   a = 10;
   b = (a == 1) ? 20: 30;
   printf( "b 的值是 %d\n", b );

   b = (a == 10) ? 20: 30;
   printf( "b 的值是 %d\n", b );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值是 4
*ptr 是 4
b 的值是 30
b 的值是 20
```

C 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 2，在这里，*x* 被赋值为 *13*，而不是 *20*，因为运算符 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	() [] -> . ++ - -	从左到右
一元	+ - ! ~ ++ - - (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

实例

请看下面的实例，了解 C 语言中运算符的优先级：

```
#include <stdio.h>

main()
{
   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;

   e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   printf("(a + b) * c / d 的值是 %d\n",  e );

   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   printf("((a + b) * c) / d 的值是 %d\n" ,  e );

   e = (a + b) * (c / d);   // (30) * (15/5)
   printf("(a + b) * (c / d) 的值是 %d\n",  e );

   e = a + (b * c) / d;     //  20 + (150/5)
   printf("a + (b * c) / d 的值是 %d\n" ,  e );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

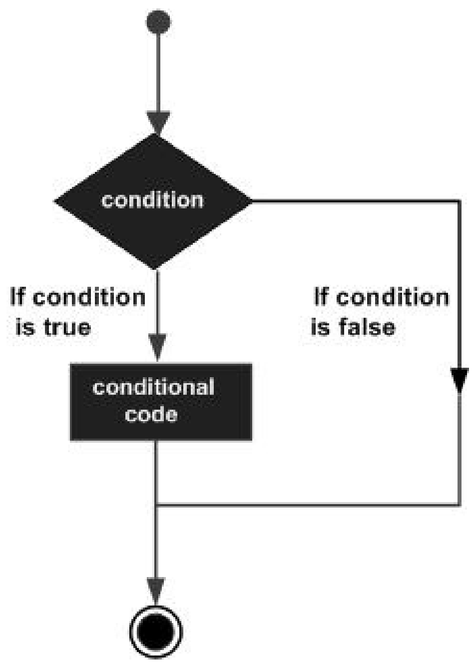
```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

C 判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

C 语言把任何非零和非空的值假定为 **true**，把零或 **null** 假定为 **false**。

下面是大多数编程语言中典型的判断结构的一般形式：



判断语句

C 语言提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
if 语句	一个 **if 语句** 由一个布尔表达式后跟一个或多个语句组成。
if...else 语句	一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。
嵌套 if 语句	您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。
switch 语句	一个 **switch** 语句允许测试一个变量等于多个值时的情况。
嵌套 switch 语句	您可以在一个 **switch** 语句内使用另一个 **switch** 语句。

? : 运算符

我们已经在前面的章节中讲解了 条件运算符 **? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

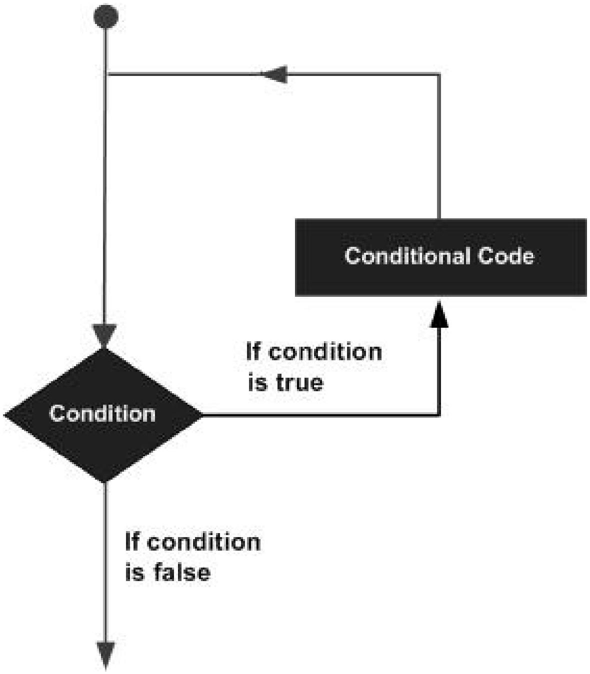
? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

C 循环

有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：



循环类型

C 语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 while、for 或 do..while 循环内使用一个或多个循环。

循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C 提供了下列的控制语句。点击链接查看每个语句的细节。

控制语句	描述
break 语句	终止 **loop** 或 **switch** 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。
continue 语句	引起循环跳过主体的剩余部分，立即重新开始测试条件。
goto 语句	将控制转移到被标记的语句。但是不建议在程序中使用 goto 语句。

无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

```
#include <stdio.h>

int main ()
{

   for( ; ; )
   {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，C 程序员偏向于使用 for(;;) 结构来表示一个无限循环。

注意：您可以按 Ctrl + C 键终止一个无限循环。

C 函数

函数是一组一起执行任务的语句。每个 C 程序都至少有一个函数，即主函数 **main()** ，所有简单的程序都可以定义其他额外的函数。

您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

C 标准库提供了大量的程序可以调用的内置函数。例如，函数 **strcat()** 用来连接两个字符串，函数 **memcpy()** 用来复制内存到另一个位置。

函数还有很多叫法，比如方法、子例程或程序，等等。

定义函数

C 语言中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
   body of the function
}
```

在 C 语言中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

- 返回类型：一个函数可以返回一个值。**return_type** 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，return_type 是关键字 **void**。
- 函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
- 参数：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
- 函数主体：函数主体包含一组定义函数执行任务的语句。

实例

以下是 **max()** 函数的源代码。该函数有两个参数 num1 和 num2，会返回这两个数中较大的那个数：

```
/* 函数返回两个数中较大的那个数 */
int max(int num1, int num2)
{
   /* 局部变量声明 */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

函数声明

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

```
return_type function_name( parameter list );
```

针对上面定义的函数 max()，以下是函数声明：

```
int max(int num1, int num2);
```

在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：

```
int max(int, int);
```

当您在一个源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

调用函数

创建 C 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。

当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。例如：

```
#include <stdio.h>

/* 函数声明 */
int max(int num1, int num2);

int main ()
{
   /* 局部变量定义 */
   int a = 100;
   int b = 200;
   int ret;

   /* 调用函数来获取最大值 */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* 函数返回两个数中较大的那个数 */
int max(int num1, int num2)
{
   /* 局部变量声明 */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

把 max() 函数和 main() 函数放一块，编译源代码。当运行最后的可执行文件时，会产生下列结果：

```
Max value is : 200
```

函数参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数。

形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有两种向函数传递参数的方式：

调用类型	描述
传值调用	该方法把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数不会影响实际参数。
引用调用	该方法把参数的地址复制给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

默认情况下，C 使用传值调用来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的实际参数。

C 作用域规则

任何一种编程中，作用域是程序中定义的变量所存在的区域，超过该区域变量就不能被访问。C 语言中有三个地方可以声明变量：

1. 在函数或块内部的局部变量
2. 在所有函数外部的全局变量
3. 在形式参数的函数参数定义中

让我们来看看什么是局部变量、全局变量和形式参数。

局部变量

在某个函数或块的内部声明的变量称为局部变量。它们只能被该函数或该代码块内部的语句使用。局部变量在函数外部是不可知的。下面是使用局部变量的实例。在这里，所有的变量 a、b 和 c 是 main() 函数的局部变量。

```
#include <stdio.h>

int main ()
{
  /* 局部变量声明 */
  int a, b;
  int c;

  /* 实际初始化 */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

全局变量

全局变量是定义在函数外部，通常是在程序的顶部。全局变量在整个程序生命周期内都是有效的，在任意的函数内部能访问全局变量。

全局变量可以被任何函数访问。也就是说，全局变量在声明后整个程序中都是可用的。下面是使用全局变量和局部变量的实例：

```
#include <stdio.h>

/* 全局变量声明 */
int g;

int main ()
{
  /* 局部变量声明 */
  int a, b;

  /* 实际初始化 */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

在程序中，局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值。下面是一个实例：

```
#include <stdio.h>

/* 全局变量声明 */
int g = 20;

int main ()
{
  /* 局部变量声明 */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of g = 10
```

形式参数

函数的参数，形式参数，被当作该函数内的局部变量，它们会优先覆盖全局变量。下面是一个实例：

```
#include <stdio.h>

/* 全局变量声明 */
int a = 20;

int main ()
{
  /* 在主函数中的局部变量声明 */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* 添加两个整数的函数 */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

初始化局部变量和全局变量

当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动对其初始化，如下所示：

数据类型	初始化默认值
int	0
char	'\0'
float	0
double	0
pointer	NULL

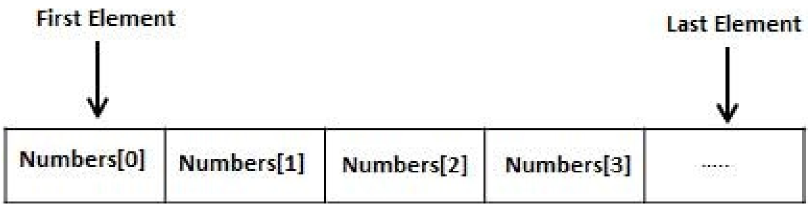
正确地初始化变量是一个良好的编程习惯，否则程序可能会产生意外的结果，因为未初始化的变量会导致一些在内存位置中已经可用的垃圾值。

C 数组

C 语言支持数组数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。

数组的声明并不是声明一个个单独的变量，比如 number0、number1、...、number99，而是声明一个数组变量，比如 numbers，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来代表一个个单独的变量。数组中的特定元素可以通过索引访问。

所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素。



声明数组

在 C 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示：

```
type arrayName [ arraySize ];
```

这叫做一维数组。**arraySize** 必须是一个大于零的整数常量，**type** 可以是任意有效的 C 数据类型。例如，要声明一个类型为 double 的包含 10 个元素的数组 **balance**，声明语句如下：

```
double balance[10];
```

现在 *balance* 是一个可用的数组，可以容纳 10 个类型为 double 的数字。

初始化数组

在 C 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

大括号 { } 之间的值的数目不能大于我们在数组声明时在方括号 [ ] 中指定的元素数目。

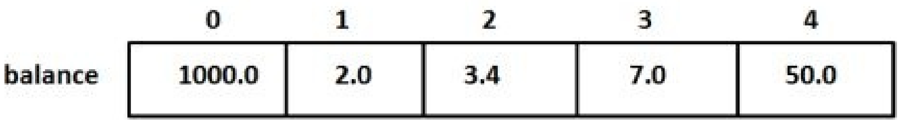
如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数。因此，如果：

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

您将创建一个数组，它与前一个实例中所创建的数组是完全相同的。下面是一个为数组中某个元素赋值的实例：

```
balance[4] = 50.0;
```

上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，数组的最后一个索引是数组的总大小减去 1。以下是上面所讨论的数组的的图形表示：



访问数组元素

数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内，跟在数组名称的后边。例如：

```
double salary = balance[9];
```

上面的语句将把数组中第 10 个元素的值赋给 salary 变量。下面的实例使用了上述的三个概念，即，声明数组、数组赋值、访问数组：

```
#include <stdio.h>

int main ()
{
   int n[ 10 ]; /* n 是一个包含 10 个整数的数组 */
   int i,j;

   /* 初始化数组元素 */
   for ( i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; /* 设置元素 i 为 i + 100 */
   }

   /* 输出数组中每个元素的值 */
   for (j = 0; j < 10; j++ )
   {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C 中数组详解

在 C 中，数组是非常重要的，且需要了解更多的细节。下面列出了 C 程序员必须清楚的一些与数组相关的重要概念：

概念	描述
多维数组	C 支持多维数组。多维数组最简单的形式是二维数组。
传递数组给函数	您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
从函数返回数组	C 允许从函数返回数组。
指向数组的指针	您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。

C 指针

学习 C 语言的指针既简单又有趣。通过指针，可以简化一些 C 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。所以，想要成为一名优秀的 C 程序员，学习指针是很有必要的。

正如您所知道的，每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。考虑下面的实例，它将输出定义的变量地址：

```
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

通过上面的实例，我们了解了什么是内存地址以及如何访问它。接下来让我们看看什么是指针。

什么是指针？

指针是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

```
type *var-name;
```

在这里，**type** 是指针的基类型，它必须是一个有效的 C 数据类型，**var-name** 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

```
int    *ip;    /* 一个整型的指针 */
double *dp;    /* 一个 double 型的指针 */
float  *fp;    /* 一个浮点型的指针 */
char   *ch     /* 一个字符型的指针 */
```

所有指针的值的实际数据类型，不管是整行、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型。

如何使用指针？

使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值。下面的实例使用了这些操作：

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* 实际变量的声明 */
   int  *ip;        /* 指针变量的声明 */

   ip = &var;  /* 在指针变量中存储 var 的地址 */

   printf("Address of var variable: %x\n", &var  );

   /* 在指针变量中存储的地址 */
   printf("Address stored in ip variable: %x\n", ip );

   /* 使用指针访问值 */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

C 中的 NULL 指针

在变量声明的时候，为指针变量赋一个 NULL 值是一个良好的编程习惯，以防没有确切的地址可以赋值的情况。赋为 NULL 值的指针被称为空指针。

NULL 指针是一个定义在标准库中的值为零的常量。请看下面的程序：

```
#include <stdio.h>

int main ()
{
   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The value of ptr is 0
```

在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，这表明 NULL 指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

如需检查一个空指针，您可以使用 if 语句，如下所示：

```
if(ptr)     /* 如果 p 非空，则完成 */
if(!ptr)    /* 如果 p 为空，则完成 */
```

C 指针详解

在 C 中，有很多指针相关的概念，这些概念都很简单，但是都很重要。下面列出了 C 程序员必须清楚的一些与指针相关的重要概念：

概念	描述
指针的算术运算	可以对指针进行四种算术运算：++、--、+、-
指针数组	可以定义用来存储指针的数组。
指向指针的指针	C 允许指向指针的指针。
传递指针给函数	通过引用或地址传递参数能使传递的参数在调用函数中被改变。
从函数返回指针	C 允许函数返回指针到局部变量、静态变量和动态内存分配。

C 字符串

在 C 语言中，字符串实际上是使用 **null** 字符 '\0' 终止的一维字符数组。因此，一个以 null 结尾的字符串，包含了组成字符串的字符。

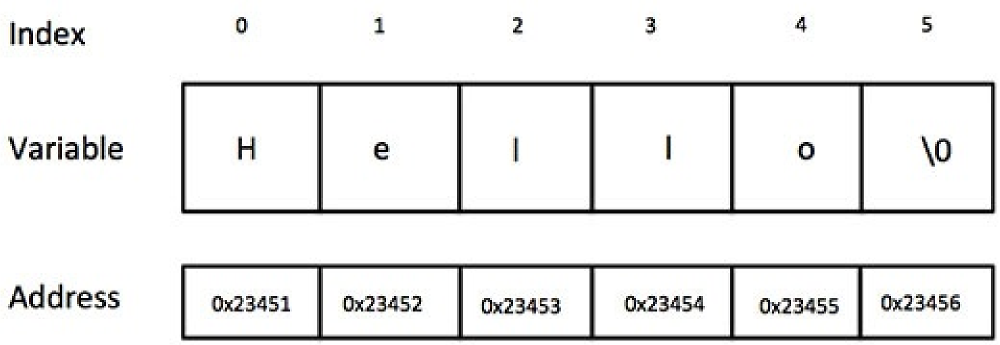
下面的声明和初始化创建了一个 "Hello" 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 "Hello" 的字符数多一个。

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

依据数组初始化规则，您可以把上面的语句写成以下语句：

```
char greeting[] = "Hello";
```

以下是 C/C++ 中定义的字符串的内存表示：



其实，您不需要把 *null* 字符放在字符串常量的末尾。C 编译器会在初始化数组时，自动把 '\0' 放在字符串的末尾。让我们尝试输出上面的字符串：

```
#include <stdio.h>

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   printf("Greeting message: %s\n", greeting );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Greeting message: Hello
```

C 中有大量操作字符串的函数：

函数	目的
strcpy(s1, s2);	复制字符串 s2 到字符串 s1。
strcat(s1, s2);	链接字符串 s2 到字符串 s1 的末尾。
strlen(s1);	返回字符串 s1 的长度。
strcmp(s1, s2);	如果,s1 和 s2 是相同的，则返回 0；如果 s1s2 则返回大于 0。
strchr(s1, ch);	返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。
strstr(s1, s2);	返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

下面的实例使用了上述的一些函数：

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* 复制 str1 到 str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* 连接 str1 和 str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* 连接后，str1 的总长度 */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

您可以在 C 标准库中找到更多字符串相关的函数。

C 结构体

C 数组允许定义可存储相同类型数据项的变量的类型，结构是 C 编程中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。

结构用于表示一条记录，假设您想要跟踪图书馆中书本的动态，您可能需要跟踪每本书的下列属性：

- Title
- Author
- Subject
- Book ID

定义结构

为了定义结构，您必须使用 **struct** 语句。struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下：

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

structure tag 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在结构定义的末尾，最后一个分号之前，您可以指定一个或多个结构变量，这是可选的。下面是声明 Book 结构的方式：

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

访问结构成员

为了访问结构的成员，我们使用成员访问运算符（**.**）。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。您可以使用 **struct** 关键字来定义结构类型的变量。下面的实例演示了结构的用法：

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Book */
   struct Books Book2;        /* 声明 Book2，类型为 Book */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 输出 Book1 信息 */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* 输出 Book2 信息 */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

结构作为函数参数

您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。您可以使用上面实例中的方式来访问结构变量：

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* 函数声明 */
void printBook( struct Books book );
int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Book */
   struct Books Book2;        /* 声明 Book2，类型为 Book */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 输出 Book1 信息 */
   printBook( Book1 );

   /* 输出 Book2 信息 */
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

指向结构的指针

您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

```
struct Books *struct_pointer;
```

现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：

```
struct_pointer = &Book1;
```

为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：

```
struct_pointer->title;
```

让我们使用结构指针来重写上面的实例，这将有助于您理解结构指针的概念：

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* 函数声明 */
void printBook( struct Books *book );
int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Book */
   struct Books Book2;        /* 声明 Book2，类型为 Book */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 通过传 Book1 的地址来输出 Book1 信息 */
   printBook( &Book1 );

   /* 通过传 Book2 的地址来输出 Book2 信息 */
   printBook( &Book2 );

   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位二进位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为"位域"或"位段"。

所谓"位域"是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

典型的实例：

- 用 1 位二进位存放一个开关量时，只有 0 和 1 两种状态。
- 读取外部文件格式——可以读取非标准的文件格式。例如：9 位的整数。

位域的定义和位域变量的说明

位域定义与结构定义相仿，其形式为：

```
    struct 位域结构名
        { 位域列表 };
```

其中位域列表的形式为：

```
    类型说明符 位域名: 位域长度
```

例如：

```
struct bs{
    int a:8;
    int b:2;
    int c:6;
};
```

位域变量的说明与结构变量说明的方式相同。 可采用先定义后说明，同时定义说明或者直接说明这三种方式。例如：

```
struct bs{
    int a:8;
    int b:2;
    int c:6;
}data;
```

说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。

让我们再来看一个实例：

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} pack;
```

在这里，packed_struct 包含了 6 个成员：四个 1 位的标识符 f1..f4、一个 4 位的 type 和一个 9 位的 my_int。

对于位域的定义尚有以下几点说明：

- 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

  ```
  struct bs{
      unsigned a:4;
      unsigned  :4;    /* 空域 */
      unsigned b:4;    /* 从下一单元开始存放 */
      unsigned c:4
  }
  ```

  在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

- 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过8位二进位。如果最大长度大于计算机的整数字长，一些编译器可能会允许域的内存重叠，另外一些编译器可能会把大于一个域的部分存储在下一个字中。

- 位域可以是无名位域，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

  ```
  struct k{
      int a:1;
      int  :2;    /* 该 2 位不能使用 */
      int b:3;
      int c:2;
  };
  ```

从以上分析可以看出，位域在本质上就是一种结构类型，不过其成员是按二进位分配的。

位域的使用

位域的使用和结构成员的使用相同，其一般形式为：

```
位域变量名·位域名
```

位域允许用各种格式输出。

请看下面的实例：

```
main(){
    struct bs{
        unsigned a:1;
        unsigned b:3;
        unsigned c:4;
    } bit,*pbit;
    bit.a=1;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    bit.b=7;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    bit.c=15;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    printf("%d,%d,%d\n",bit.a,bit.b,bit.c);    /* 以整型量格式输出三个域的内容 */
    pbit=&bit;    /* 把位域变量 bit 的地址送给指针变量 pbit */
    pbit->a=0;    /* 用指针方式给位域 a 重新赋值，赋为 0 */
    pbit->b&=3;    /* 使用了复合的位运算符 "&="，相当于：pbit->b=pbit->b&3，位域 b 中原有值为 7，
    pbit->c|=1;    /* 使用了复合位运算符"|="，相当于：pbit->c=pbit->c|1，其结果为 15 */
    printf("%d,%d,%d\n",pbit->a,pbit->b,pbit->c);    /* 用指针方式输出了这三个域的值 */
}
```

上例程序中定义了位域结构 bs，三个位域为 a、b、c。说明了 bs 类型的变量 bit 和指向 bs 类型的指针变量 pbit。这表示位域也是可以使用指针的。

C 共用体

共用体是一种特殊的数据类型，允许您在相同的内存位置存储不同的数据类型。您可以定义一个带有多成员的共用体，但是任何时候只能有一个成员带有值。共用体提供了一种使用相同的内存位置的有效方式。

定义共用体

为了定义结构体，您必须使用 **union** 语句，方式与定义结构类似。union 语句定义了一个新的数据类型，带有多个成员。union 语句的格式如下：

```
union [union tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

union tag 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在共用体定义的末尾，最后一个分号之前，您可以指定一个或多个共用体变量，这是可选的。下面定义一个名为 Data 的共用体类型，有三个成员 i、f 和 str：

```
union Data
{
   int i;
   float f;
   char  str[20];
} data;
```

现在，**Data** 类型的变量可以存储一个整数、一个浮点数，或者一个字符串。这意味着一个变量（相同的内存位置）可以存储多个多种类型的数据。您可以根据需要在一个共用体内使用任何内置的或者用户自定义的数据类型。

共用体占用的内存应足够存储共用体中最大的成员。例如，在上面的实例中，Data 将占用 20 个字节的内存空间，因为在各个成员中，字符串所占用的空间是最大的。下面的实例将显示上面的共用体占用的总内存大小：

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Memory size occupied by data : 20
```

访问共用体成员

为了访问共用体的成员，我们使用成员访问运算符（.）。成员访问运算符是共用体变量名称和我们要访问的共用体成员之间的一个句号。您可以使用 **union** 关键字来定义共用体类型的变量。下面的实例演示了共用体的用法：

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

在这里，我们可以看到共用体的 **i** 和 **f** 成员的值有损坏，因为最后赋给变量的值占用了内存位置，这也是 **str** 成员能够完好输出的原因。现在让我们再来看一个相同的实例，这次我们在同一时间只使用一个变量，这也演示了使用共用体的主要目的：

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

在这里，所有的成员都能完好输出，因为同一时间只用到一个成员。

C 位域

如果程序的结构中包含多个开关量，只有 TRUE/FALSE 变量，如下：

```
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

这种结构需要 8 字节的内存空间，但在实际上，在每个变量中，我们只存储 0 或 1。在这种情况下，C 语言提供了一中更好的利用内存空间的方式。如果您在结构内使用这样的变量，您可以定义变量的宽度来告诉编译器，您将只使用这些字节。例如，上面的结构可以重写成：

```
struct
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

现在，上面的结构中，status 变量将占用 4 个字节的内存空间，但是只有 2 位被用来存储值。如果您用了 32 个变量，每一个变量宽度为 1 位，那么 status 结构将使用 4 个字节，但只要您再多用一个变量，如果使用了 33 个变量，那么它将分配内存的下一段来存储第 33 个变量，这个时候就开始使用 8 个字节。让我们看看下面的实例来理解这个概念：

```
#include <stdio.h>
#include <string.h>

/* 定义简单的结构 */
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;

/* 定义位域结构 */
struct
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status2;

int main( )
{
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

位域声明

在结构内声明位域的形式如下：

```
struct
{
  type [member_name] : width ;
};
```

下面是有关位域中变量元素的描述：

元素	描述
type	整数类型，决定了如何解释位域的值。类型可以是整型、有符号整型、无符号整型。
member_name	位域的名称。
width	位域中位的数量。宽度必须小于或等于指定类型的位宽度。

带有预定义宽度的变量被称为位域。位域可以存储多于 1 位的数，例如，需要一个变量来存储从 0 到 7 的值，您可以定义一个宽度为 3 位的位域，如下：

```
struct
{
  unsigned int age : 3;
} Age;
```

上面的结构定义指示 C 编译器，age 变量将只使用 3 位来存储这个值，如果您试图使用超过 3 位，则无法完成。让我们来看下面的实例：

```
#include <stdio.h>
#include <string.h>

struct
{
  unsigned int age : 3;
} Age;

int main( )
{
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );

   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );

   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );

   return 0;
}
```

当上面的代码被编译时，它会带有警告，当上面的代码被执行时，它会产生下列结果：

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

C typedef

C 语言提供了 **typedef** 关键字，您可以使用它来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 **BYTE**：

```
typedef unsigned char BYTE;
```

在这个类型定义之后，标识符 BYTE 可作为类型 **unsigned char** 的缩写，例如：

```
BYTE  b1, b2;
```

按照惯例，定义时会大写字母，以便提醒用户类型名称是一个象征性的缩写，但您也可以使用小写字母，如下：

```
typedef unsigned char byte;
```

您也可以使用 **typedef** 来为用户自定义的数据类型取一个新的名字。例如，您可以对结构体使用 typedef 来定义一个新的数据类型，然后使用这个新的数据类型来直接定义结构变量，如下：

```
#include <stdio.h>
#include <string.h>

typedef struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Book;

int main( )
{
   Book book;

   strcpy( book.title, "C Programming");
   strcpy( book.author, "Nuha Ali");
   strcpy( book.subject, "C Programming Tutorial");
   book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book  title : C Programming
Book  author : Nuha Ali
Book  subject : C Programming Tutorial
Book  book_id : 6495407
```

typedef vs #define

#define 是 C 指令，用于为各种数据类型定义别名，与 **typedef** 类似，但是它们有以下几点不同：

- **typedef** 仅限于为类型定义符号名称，**#define** 不仅可以为类型定义别名，也能为数值定义别名，比如您可以定义 1 为 ONE。
- **typedef** 是由编译器执行解释的，**#define** 语句是由预编译器进行处理的。

下面是 #define 的最简单的用法：

```
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int main( )
{
   printf( "Value of TRUE : %d\n", TRUE);
   printf( "Value of FALSE : %d\n", FALSE);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of TRUE : 1
Value of FALSE : 0
```

C 输入 & 输出

当我们提到输入时，这意味着要向程序填充一些数据。输入可以是以文件的形式或从命令行中进行。C 语言提供了一系列内置的函数来读取给定的输入，并根据需要填充到程序中。

当我们提到输出时，这意味着要在屏幕上、打印机上或任意文件中显示一些数据。C 语言提供了一系列内置的函数来输出数据到计算机屏幕上和保存数据到文本文件或二进制文件中。

标准文件

C 语言把所有的设备都当作文件。所以设备（比如显示器）被处理的方式与文件相同。以下三个文件会在程序执行时自动打开，以便访问键盘和屏幕。

标准文件	文件指针	设备
标准输入	stdin	键盘
标准输出	stdout	屏幕
标准错误	stderr	您的屏幕

文件指针是访问文件的方式，本节将讲解如何从屏幕读取值以及如何把结果输出到屏幕上。

getchar() & putchar() 函数

int getchar(void) 函数从屏幕读取下一个可用的字符，并把它返回为一个整数。这个函数在同一个时间内只会读取一个单一的字符。您可以在循环内使用这个方法，以便从屏幕上读取多个字符。

int putchar(int c) 函数把字符输出到屏幕上，并返回相同的字符。这个函数在同一个时间内只会输出一个单一的字符。您可以在循环内使用这个方法，以便在屏幕上输出多个字符。

请看下面的实例：

```
#include <stdio.h>
int main( )
{
   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并只会读取一个单一的字符，显示如下：

```
$./a.out
<b>Enter a value :</b> this is test
<b>You entered:</b> t
```

gets() & puts() 函数

char *gets(char* s) 函数从 **stdin** 读取一行到 **s** 所指向的缓冲区，直到一个终止符或 EOF。

int puts(const char *s) 函数把字符串 s 和一个尾随的换行符写入到 **stdout**。

```
#include <stdio.h>
int main( )
{
   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取一整行直到该行结束，显示如下：

```
$./a.out
<b>Enter a value :</b> this is test
<b>You entered:</b> This is test
```

scanf() 和 printf() 函数

int scanf(const char *format, ...) 函数从标准输入流 **stdin** 读取输入，并根据提供的 **format** 来浏览输入。

int printf(const char *format, ...) 函数把输出写入到标准输出流 **stdout** ，并根据提供的格式产生输出。

format 可以是一个简单的常量字符串，但是您可以分别指定 %s、%d、%c、%f 等来输出或读取字符串、整数、字符或浮点数。还有许多其他可用的格式选项，可以根据需要使用。如需了解完整的细节，可以查看这些函数的参考手册。现在让我们通过下面这个简单的实例来加深理解：

```
#include <stdio.h>
int main( )
{
   char str[100];
   int i;

   printf( "Enter a value :");
   scanf("%s %d", str, &i);

   printf( "\nYou entered: %s %d ", str, i);

   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取输入，显示如下：

```
$./a.out
<b>Enter a value :</b> seven 7
<b>You entered:</b> seven 7
```

在这里，应当指出的是，scanf() 期待输入的格式与您给出的 %s 和 %d 相同，这意味着您必须提供有效的输入，比如 "string integer"，如果您提供的是 "string string" 或 "integer integer"，它会被认为是错误的输入。另外，在读取字符串时，只要遇到一个空格，scanf() 就会停止读取，所以 "this is test" 对 scanf() 来说是三个字符串。

C 文件读写

上一章我们讲解了 C 语言处理的标准输入和输出设备。本章我们将介绍 C 程序员如何创建、打开、关闭文本文件或二进制文件。

一个文件，无论它是文本文件还是二进制文件，都是代表了一系列的字节。C 语言不仅提供了访问顶层的函数，也提供了底层（OS）调用来处理存储设备上的文件。本章将讲解文件管理的重要调用。

打开文件

您可以使用 **fopen( )** 函数来创建一个新的文件或者打开一个已有的文件，这个调用会初始化类型 **FILE** 的一个对象，类型 **FILE** 包含了所有用来控制流的必要的信息。下面是这个函数调用的原型：

```
FILE *fopen( const char * filename, const char * mode );
```

在这里，**filename** 是字符串，用来命名文件，访问模式 **mode** 的值可以是下列值中的一个：

模式	描述
r	打开一个已有的文本文件，允许读取文件。
w	打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。
a	打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。
r+	打开一个文本文件，允许读写文件。
w+	打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。
a+	打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。

如果处理的是二进制文件，则需使用下面的访问模式来取代上面的访问模式：

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

关闭文件

；

为了关闭文件，请使用 fclose( ) 函数。函数的原型如下：

```
int fclose( FILE *fp );
```

如果成功关闭文件，**fclose( )** 函数返回零，如果关闭文件时发生错误，函数返回 **EOF**。这个函数实际上，会清空缓冲区中的数据，关闭文件，并释放用于该文件的所有内存。EOF 是一个定义在头文件 **stdio.h** 中的常量。

C 标准库提供了各种函数来按字符或者以固定长度字符串的形式读写文件。

写入文件

下面是把字符写入到流中的最简单的函数：

```
int fputc( int c, FILE *fp );
```

函数 **fputc()** 把参数 c 的字符值写入到 fp 所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 **EOF**。您可以使用下面的函数来把一个以 null 结尾的字符串写入到流中：

```
int fputs( const char *s, FILE *fp );
```

函数 **fputs()** 把字符串 **s** 写入到 fp 所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回 **EOF**。您也可以使用 **int fprintf(FILE *fp,const char* format, ...)** 函数来写把一个字符串写入到文件中。尝试下面的实例：

注意：请确保您有可用的 **/tmp** 目录，如果不存在该目录，则需要在您的计算机上先创建该目录。

```
#include <stdio.h>

main()
{
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

当上面的代码被编译和执行时，它会在 /tmp 目录中创建一个新的文件 **test.txt**，并使用两个不同的函数写入两行。接下来让我们来读取这个文件。

读取文件

下面是从文件读取单个字符的最简单的函数：

```
int fgetc( FILE * fp );
```

fgetc() 函数从 fp 所指向的输入文件中读取一个字符。返回值是读取的字符，如果发生错误则返回 **EOF**。下面的函数允许您从流中读取一个字符串：

```
char *fgets( char *buf, int n, FILE *fp );
```

函数 **fgets()** 从 fp 所指向的输入流中读取 n - 1 个字符。它会把读取的字符串复制到缓冲区 **buf**，并在最后追加一个 **null** 字符来终止字符串。

如果这个函数在读取最后一个字符之前就遇到一个换行符 '\n' 或文件的末尾 EOF，则只会返回读取到的字符，包括换行符。您也可以使用 **int fscanf(FILE *fp, const char* format, ...)** 函数来从文件中读取字符串，但是在遇到第一个空格字符时，它会停止读取。

```
#include <stdio.h>

main()
{
   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

当上面的代码被编译和执行时，它会读取上一部分创建的文件，产生下列结果：

```
1 : This
2: is testing for fprintf...

3: This is testing for fputs...
```

首先，**fscanf()** 方法只读取了 **This**，因为它在后边遇到了一个空格。其次，调用 **fgets()** 读取剩余的部分，直到行尾。最后，调用 **fgets()** 完整地读取第二行。

二进制 I/O 函数

下面两个函数用于二进制输入和输出：

```
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

这两个函数都是用于存储块的读写 - 通常是数组或结构体。

C 预处理器

C 预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器只不过是一个文本替换工具而已，他们会指示编译器在实际编译之前完成所需的预处理。我们将把 C 预处理器（C Preprocessor）简写为 CPP。

所有的预处理器命令都是以井号（#）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一列开始。下面列出了所有重要的预处理器指令：

指令	描述
#define	定义宏
#include	包含一个源代码文件
#undef	取消已定义的宏
#ifdef	如果宏已经定义，则返回真
#ifndef	如果宏没有定义，则返回真
#if	如果给定条件为真，则编译下面代码
#else	#if 的替代方案
#elif	如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个 #if……#else 条件编译块
#error	当遇到标准错误时，输出错误消息
#pragma	使用标准化方法，向编译器发布特殊的命令到编译器中

预处理器实例

分析下面的实例来理解不同的指令。

```
#define MAX_ARRAY_LENGTH 20
```

这个指令告诉 CPP 把所有的 MAX*ARRAY_LENGTH* 替换为 *20*。使用 #define_ 定义常量来增强可读性。

```
#include <stdio.h>
#include "myheader.h"
```

这些指令告诉 CPP 从系统库中获取 stdio.h，并添加文本到当前的源文件中。下一行告诉 CPP 从本地目录中获取 **myheader.h**，并添加内容到当前的源文件中。

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

这个指令告诉 CPP 取消已定义的 FILE_SIZE，并定义它为 42。

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

这个指令告诉 CPP 只有当 MESSAGE 未定义时，才定义 MESSAGE。

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

这个指令告诉 CPP 如果定义了 DEBUG，则执行处理语句。在编译时，如果您向 gcc 编译器传递了 *-DDEBUG* 开关量，这个指令就非常有用。它定义了 DEBUG，您可以在编译期间随时开启或关闭调试。

预定义宏

ANSI C 定义了许多宏。在编程中您可以使用这些宏，但是不同直接修改这些预定义的宏。

宏	描述
__DATE__	当前日期，一个以 "MMM DD YYYY" 格式表示的字符常量。
__TIME__	当前时间，一个以 "HH:MM:SS" 格式表示的字符常量。
__FILE__	这会包含当前文件名，一个字符串常量。
__LINE__	这会包含当前行号，一个十进制常量。
__STDC__	当编译器以 ANSI 标准编译时，则定义为 1。

让我们来尝试下面的实例：

```
#include <stdio.h>

main()
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );

}
```

当上面的代码（在文件 **test.c** 中）被编译和执行时，它会产生下列结果：

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

预处理器运算符

C 预处理器提供了下列的运算符来帮助您创建宏：

宏延续运算符（\）

一个宏通常写在一个单行上。但是如果宏太长，一个单行容纳不下，则使用宏延续运算符（\）。例如：

```
#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")
```

字符串常量化运算符（**#**）

在宏定义中，当需要把一个宏的参数转换为字符串常量时，则使用字符串常量化运算符（#）。在宏中使用的该运算符有一个特定的参数或参数列表。例如：

```
#include <stdio.h>

#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")

int main(void)
{
   message_for(Carole, Debra);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Carole and Debra: We love you!
```

标记粘贴运算符（**##**）

宏定义内的标记粘贴运算符（##）会合并两个参数。它允许在宏定义中两个独立的标记被合并为一个标记。例如：

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)
{
   int token34 = 40;

   tokenpaster(34);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
token34 = 40
```

这是怎么发生的，因为这个实例会从编译器产生下列的实际输出：

```
printf ("token34 = %d", token34);
```

这个实例演示了 token##n 会连接到 token34 中，在这里，我们使用了字符串常量化运算符（**#**）和标记粘贴运算符（**##**）。

defined() 运算符

预处理器 **defined** 运算符是用在常量表达式中的，用来确定一个标识符是否已经使用 #define 定义过。如果指定的标识符已定义，则值为真（非零）。如果指定的标识符未定义，则值为假（零）。下面的实例演示了 defined() 运算符的用法：

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void)
{
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Here is the message: You wish!
```

参数化的宏

CPP 一个强大的功能是可以使用参数化的宏来模拟函数。例如，下面的代码是计算一个数的平方：

```
int square(int x) {
   return x * x;
}
```

我们可以使用宏重写上面的代码，如下：

```
#define square(x) ((x) * (x))
```

在使用带有参数的宏之前，必须使用 **#define** 指令定义。参数列表是括在圆括号内，且必须紧跟在宏名称的后边。宏名称和左圆括号之间不允许有空格。例如：

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max between 20 and 10 is 20
```

C 头文件

头文件是扩展名为 **.h** 的文件，包含了 C 函数声明和宏定义，被多个源文件中引用共享。有两种类型的头文件：程序员编写的头文件和编译器自带的头文件。

在程序中要使用头文件，需要使用 C 预处理指令 **#include** 来引用它。前面我们已经看过 **stdio.h** 头文件，它是编译器自带的头文件。

引用头文件相当于复制头文件的内容，但是我们不会直接在源文件中复制头文件的内容，因为这么做很容易出错，特别在程序是由多个源文件组成的时候。

A simple practice in C 或 C++ 程序中，建议把所有的常量、宏、系统全局变量和函数原型写在头文件中，在需要的时候随时引用这些头文件。

引用头文件的语法

使用预处理指令 **#include** 可以引用用户和系统头文件。它的形式有以下两种：

```
#include <file>
```

这种形式用于引用系统头文件。它在系统目录的标准列表中搜索名为 file 的文件。在编译源代码时，您可以通过 -I 选项把目录前置在该列表前。

```
#include "file"
```

这种形式用于引用用户头文件。它在包含当前文件的目录中搜索名为 file 的文件。在编译源代码时，您可以通过 -I 选项把目录前置在该列表前。

引用头文件的操作

#include 指令会指示 C 预处理器浏览指定的文件作为输入。预处理器的输出包含了已经生成的输出，被引用文件生成的输出以及 **#include** 指令之后的文本输出。例如，如果您有一个头文件 header.h，如下：

```
char *test (void);
```

和一个使用了头文件的主程序 *program.c*，如下：

```
int x;
#include "header.h"

int main (void)
{
   puts (test ());
}
```

编译器会看到如下的令牌流：

```
int x;
char *test (void);

int main (void)
{
   puts (test ());
}
```

只引用一次头文件

如果一个头文件被引用两次，编译器会处理两次头文件的内容，这将产生错误。为了防止这种情况，标准的做法是把文件的整个内容放在条件编译语句中，如下：

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

这种结构就是通常所说的包装器 **#ifndef**。当再次引用头文件时，条件为假，因为 HEADER_FILE 已定义。此时，预处理器会跳过文件的整个内容，编译器会忽略它。

有条件引用

有时需要从多个不同的头文件中选择一个引用到程序中。例如，需要指定在不同的操作系统上使用的配置参数。您可以通过一系列条件来实现这点，如下：

```
#if SYSTEM_1
   # include "system_1.h"
#elif SYSTEM_2
   # include "system_2.h"
#elif SYSTEM_3
   ...
#endif
```

但是如果头文件比较多的时候，这么做是很不妥当的，预处理器使用宏来定义头文件的名称。这就是所谓的有条件引用。它不是用头文件的名称作为 **#include** 的直接参数，您只需要使用宏名称代替即可：

```
#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H
```

SYSTEM_H 会扩展，预处理器会查找 system_1.h，就像 **#include** 最初编写的那样。SYSTEM_H 可通过 -D 选项被您的 Makefile 定义。

C 强制类型转换

强制类型转换是把变量从一种类型转换为另一种数据类型。例如，如果您想存储一个 long 类型的值到一个简单的整型中，您需要把 long 类型强制转换为 int 类型。您可以使用**强**制类型转换运算符来把值显式地从一种类型转换为另一种类型，如下所示：

```
(type_name) expression
```

请看下面的实例，使用强制类型转换运算符把一个整数变量除以另一个整数变量，得到一个浮点数：

```
#include <stdio.h>

main()
{
   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of mean : 3.400000
```

这里要注意的是强制类型转换运算符的优先级大于除法，因此 **sum** 的值首先被转换为 **double** 型，然后除以 count，得到一个类型为 double 的值。

类型转换可以是隐式的，由编译器自动执行，也可以是显式的，通过使用**强**制类型转换运算符来指定。在编程时，有需要类型转换的时候都用上强制类型转换运算符，是一种良好的编程习惯。

整数提升

整数提升是指把小于 **int** 或 **unsigned int** 的整数类型转换为 **int** 或 **unsigned int** 的过程。请看下面的实例，在 int 中添加一个字符：

```
#include <stdio.h>

main()
{
   int  i = 17;
   char c = 'c'; /* ascii 值是 99 */
   int sum;

   sum = i + c;
   printf("Value of sum : %d\n", sum );

}
```

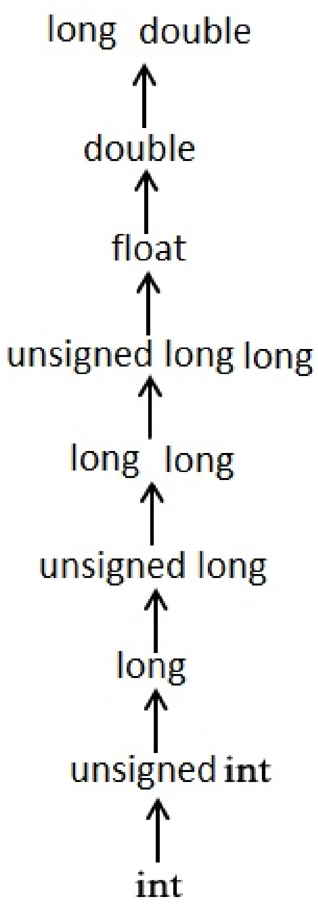
当上面的代码被编译和执行时，它会产生下列结果：

```
Value of sum : 116
```

在这里，sum 的值为 116，因为编译器进行了整数提升，在执行实际加法运算时，把 'c' 的值转换为对应的 ascii 值。

常用的算术转换

常用的算术转换是隐式地把值强制转换为相同的类型。编译器首先执行整数提升，如果操作数类型不同，则它们会被转换为下列层次中出现的最高层次的类型：



常用的算术转换不适用于赋值运算符、逻辑运算符 && 和 ||。让我们看看下面的实例来理解这个概念：

```
#include <stdio.h>

main()
{
   int  i = 17;
   char c = 'c'; /* ascii 值是 99 */
   float sum;

   sum = i + c;
   printf("Value of sum : %f\n", sum );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of sum : 116.000000
```

在这里，c 首先被转换为整数，但是由于最后的值是 double 型的，所以会应用常用的算术转换，编译器会把 i 和 c 转换为浮点型，并把它们相加得到一个浮点数。

C 错误处理

C 语言不提供对错误处理的直接支持，但是作为一种系统编程语言，它以返回值的形式允许您访问底层数据。在发生错误时，大多数的 C 或 UNIX 函数调用返回 1 或 NULL，同时会设置一个错误代码 **errno**，该错误代码是全局变量，表示在函数调用期间发生了错误。您可以在 <error.h> 头文件中找到各种各样的错误代码。

所以，C 程序员可以通过检查返回值，然后根据返回值决定采取哪种适当的动作。开发人员应该在程序初始化时，把 errno 设置为 0，这是一种良好的编程习惯。0 值表示程序中没有错误。

errno、perror() 和 strerror()

C 语言提供了 **perror()** 和 **strerror()** 函数来显示与 **errno** 相关的文本消息。

- **perror()** 函数显示您传给它的字符串，后跟一个冒号、一个空格和当前 errno 值的文本表示形式。
- **strerror()** 函数，返回一个指针，指针指向当前 errno 值的文本表示形式。

让我们来模拟一种错误情况，尝试打开一个不存在的文件。您可以使用多种方式来输出错误消息，在这里我们使用函数来演示用法。另外有一点需要注意，您应该使用 **stderr** 文件流来输出所有的错误。

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main ()
{
   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL)
   {
      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
   }
   else
   {
      fclose (pf);
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

被零除的错误

在进行除法运算时，不检查除数是否为零，这是程序员编程时常见的问题，会导致一个运行时错误。

为了避免这种情况发生，下面的代码在进行处罚运算前会先检查除数是否为零：

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int dividend = 20;
   int divisor = 0;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(-1);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(0);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Division by zero! Exiting...
```

程序退出状态

通常情况下，程序成功执行完一个操作正常退出的时候会带有值 EXIT_SUCCESS。在这里，EXIT_SUCCESS 是宏，它被定义为 0。

如果程序中存在一种错误情况，当您退出程序时，会带有状态值 EXIT_FAILURE，被定义为 -1。所以，上面的程序可以写成：

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int dividend = 20;
   int divisor = 5;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(EXIT_FAILURE);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(EXIT_SUCCESS);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of quotient : 4
```

C 递归

递归是以自相似的方式重复项目的处理过程。同样地，在编程语言中，在函数内部调用函数自身，称为递归调用。如下：

```
void recursion()
{
   recursion(); /* 函数调用自身 */
}

int main()
{
   recursion();
}
```

C 语言支持递归，即，一个函数可以调用自身。但在使用递归时，程序员需要注意定义一个从函数退出的条件，否则会进入无限循环。

递归函数在解决许多数学问题上起了至关重要的作用，比如计算一个数的阶乘、生成斐波那契数列，等等。

数的阶乘

下面的实例使用递归函数计算一个给定的数的阶乘：

```
#include <stdio.h>

int factorial(unsigned int i)
{
   if(i <= 1)
   {
      return 1;
   }
   return i * factorial(i - 1);
}
int  main()
{
    int i = 15;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Factorial of 15 is 2004310016
```

斐波那契数列

下面的实例使用递归函数生成一个给定的数的斐波那契数列：

```
#include <stdio.h>

int fibonaci(int i)
{
   if(i == 0)
   {
      return 0;
   }
   if(i == 1)
   {
      return 1;
   }
   return fibonaci(i-1) + fibonaci(i-2);
}

int  main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
       printf("%d\t%n", fibonaci(i));
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
0	1	1	2	3	5	8	13	21	34
```

C 可变参数

有时，您可能会碰到这样的情况，您希望函数带有可变数量的参数，而不是预定义数量的参数。C 语言为这种情况提供了一个解决方案，它允许您定义一个函数，能根据具体的需求接受可变数量的参数。下面的实例演示了这种函数的定义。

```
int func(int, ... )
{
   .
   .
   .
}

int main()
{
   func(1, 2, 3);
   func(1, 2, 3, 4);
}
```

请注意，函数 **func()** 最后一个参数写成省略号，即三个点号（**...**），省略号之前的那个参数总是 **int**，代表了要传递的可变参数的总数。为了使用这个功能，您需要使用 **stdarg.h** 头文件，该文件提供了实现可变参数功能的函数和宏。具体步骤如下：

- 定义一个函数，最后一个参数为省略号，省略号前面的那个参数总是 **int**，表示了参数的个数。
- 在函数定义中创建一个 **va_list** 类型变量，该类型是在 stdarg.h 头文件中定义的。
- 使用 **int** 参数和 **va_start** 宏来初始化 **va_list** 变量为一个参数列表。宏 va_start 是在 stdarg.h 头文件中定义的。
- 使用 **va_arg** 宏和 **va_list** 变量来访问参数列表中的每个项。
- 使用宏 **va_end** 来清理赋予 **va_list** 变量的内存。

现在让我们按照上面的步骤，来编写一个带有可变数量参数的函数，并返回它们的平均值：

```
#include <stdio.h>
#include <stdarg.h>

double average(int num,...)
{

    va_list valist;
    double sum = 0.0;
    int i;

    /* 为 num 个参数初始化 valist */
    va_start(valist, num);

    /* 访问所有赋给 valist 的参数 */
    for (i = 0; i < num; i++)
    {
       sum += va_arg(valist, int);
    }
    /* 清理为 valist 保留的内存 */
    va_end(valist);

    return sum/num;
}

int main()
{
   printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
   printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

当上面的代码被编译和执行时，它会产生下列结果。应该指出的是，函数 **average()** 被调用两次，每次第一个参数都是表示被传的可变参数的总数。省略号被用来传递可变数量的参数。

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

C 内存管理

本章将讲解 C 中的动态内存管理。C 语言为内存的分配和管理提供了几个函数。这些函数可以在 **<stdlib.h>** 头文件中找到。

函数	描述
void *calloc(int num, int size);	该函数分配一个带有 function allocates an array of **num** 个元素的数组，每个元素的大小为 **size** 字节。
void free(void *address);	该函数释放 address 所指向的h内存块。
void *malloc(int num);	该函数分配一个 **num** 字节的数组，并把它们进行初始化。
void *realloc(void *address, int newsize);	该函数重新分配内存，把内存扩展到 **newsize**。

动态分配内存

编程时，如果您预先知道数组的大小，那么定义数组时就比较容易。例如，一个存储人名的数组，它最多容纳 100 个字符，所以您可以定义数组，如下所示：

```
char name[100];
```

但是，如果您预先不知道需要存储的文本长度，例如您向存储有关一个主题的详细描述。在这里，我们需要定义一个指针，该指针指向未定义所学内存大小的字符，后续再根据需求来分配内存，如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* 动态分配内存 */
   description = malloc( 200 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcpy( description, "Zara ali a DPS student in class 10th");
   }
   printf("Name = %s\n", name );
   printf("Description: %s\n", description );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

上面的程序也可以使用 **calloc()** 来编写，只需要把 malloc 替换为 calloc 即可，如下所示：

```
calloc(200, sizeof(char));
```

当动态分配内存时，您有完全控制权，可以传递任何大小的值。而那些预先定义了大小的数组，一旦定义则无法改变大小。

重新调整内存的大小和释放内存

当程序退出时，操作系统会自动释放所有分配给程序的内存，但是，建议您在不需要内存时，都应该调用函数 **free()** 来释放内存。

或者，您可以通过调用函数 **realloc()** 来增加或减少已分配的内存块的大小。让我们使用 realloc() 和 free() 函数，再次查看上面的实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* 动态分配内存 */
   description = malloc( 30 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcpy( description, "Zara ali a DPS student.");
   }
   /* 假设您想要存储更大的描述信息 */
   description = realloc( description, 100 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcat( description, "She is in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* 使用 free() 函数释放内存 */
   free(description);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```

您可以尝试一下不重新分配额外的内存，strcat() 函数会生成一个错误，因为存储 description 时可用的内存不足。

C 命令行参数

执行程序时，可以从命令行传值给 C 程序。这些值被称为命令行参数，它们对程序很重要，特别是当您想从外部控制程序，而不是在代码内对这些值进行硬编码时，就显得尤为重要了。

命令行参数是使用 main() 函数参数来处理的，其中，**argc** 是指传入参数的个数，**argv[]** 是一个指针数组，指向传递给程序的每个参数。下面是一个简单的实例，检查命令行是否有提供参数，并根据参数执行相应的动作：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

使用一个参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out testing
The argument supplied is testing
```

使用两个参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out testing1 testing2
Too many arguments supplied.
```

不传任何参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out
One argument expected
```

应当指出的是，**argv[0]** 存储程序的名称，**argv[1]** 是一个指向第一个命令行参数的指针，*argv[n]* 是最后一个参数。如果没有提供任何参数，*argc* 将为 *1*，否则，如果传递了一个参数，**argc* 将被设置为 2。

多个命令行参数之间用空格分隔，但是如果参数本身带有空格，那么传递参数的时候应把参数放置在双引号 "" 或单引号 '' 内部。让我们重新编写上面的实例，有一个空间，那么你可以通过这样的观点，把它们放在双引号或单引号""""。让我们重新编写上面的实例，向程序传递一个放置在双引号内部的命令行参数：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   printf("Program name %s\n", argv[0]);

   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

使用一个用空格分隔的简单参数，参数括在双引号中，编译并执行上面的代码，它会产生下列结果：

```
$./a.out "testing1 testing2"

Progranm name ./a.out
The argument supplied is testing1 testing2
```

C语言参考

C 标准库 - <assert.h>

简介

C 标准库的 **assert.h**头文件提供了一个名为 **assert** 的宏，它可用于验证程序做出的假设，并在假设为假时输出诊断消息。

已定义的宏 **assert** 指向另一个宏 **NDEBUG**，宏 **NDEBUG** 不是 <assert.h> 的一部分。如果已在引用 <assert.h> 的源文件中定义 NDEBUG 为宏名称，则 **assert** 宏的定义如下：

```
#define assert(ignore) ((void)0)
```

库宏

下面列出了头文件 assert.h 中定义的唯一的函数：

函数	描述
void assert(int expression)	这实际上是一个宏，不是一个函数，可用于在 C 程序中添加诊断。

C 库宏 - assert()

描述

C 库宏 **void assert(int expression)** 允许诊断信息被写入到标准错误文件中。换句话说，它可用于在 C 程序中添加诊断。

声明

下面是 assert() 宏的声明。

```
void assert(int expression);
```

参数

- **expression** -- 这可以是一个变量或任何 C 表达式。如果 **expression** 为 TRUE，assert() 不执行任何动作。如果 **expression** 为 FALSE，assert() 会在标准错误 stderr 上显示错误消息，并中止程序执行。

返回值

这个宏不返回任何值。

实例

下面的实例演示了 assert() 宏的用法。

```
#include <assert.h>
#include <stdio.h>

int main()
{
   int a;
   char str[50];

   printf("请输入一个整数值： ");
   scanf("%d\n", &a);
   assert(a >= 10);
   printf("输入的整数是： %d\n", a);

   printf("请输入字符串： ");
   scanf("%s\n", &str);
   assert(str != NULL);
   printf("输入的字符串是： %s\n", str);

   return(0);
}
```

让我们在交互模式下编译并运行上面的程序，如下所示：

```
请输入一个整数值： 11
输入的整数是： 11
请输入字符串： w3cschool
输入的字符串是： w3cschool
```

C 标准库 - <ctype.h>

简介

C 标准库的 **ctype.h** 头文件提供了一些函数，可用于测试和映射字符。

这些函数接受 **int** 作为参数，它的值必须是 EOF 或表示为一个无符号字符。

如果参数 c 满足描述的条件，则这些函数返回非零（true）。如果参数 c 不满足描述的条件，则这些函数返回零。

库函数

下面列出了头文件 ctype.h 中定义的函数：

函数	描述
int isalnum(int c)	该函数检查所传的字符是否是字母和数字。
int isalpha(int c)	该函数检查所传的字符是否是字母。
int iscntrl(int c)	该函数检查所传的字符是否是控制字符。
int isdigit(int c)	该函数检查所传的字符是否是十进制数字。
int isgraph(int c)	该函数检查所传的字符是否有图形表示法。
int islower(int c)	该函数检查所传的字符是否是小写字母。
int isprint(int c)	该函数检查所传的字符是否是可打印的。
int ispunct(int c)	该函数检查所传的字符是否是标点符号字符。
int isspace(int c)	该函数检查所传的字符是否是空白字符。
int isupper(int c)	该函数检查所传的字符是否是大写字母。
int isxdigit(int c)	该函数检查所传的字符是否是十六进制数字。

标准库还包含了两个转换函数，它们接受并返回一个 "int"

函数	描述
int tolower(int c)	该函数把大写字母转换为小写字母。
int toupper(int c)	该函数把小写字母转换为大写字母。

字符类

字符类	描述
数字	完整的数字集合 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
十六进制数字	集合 { 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f }
小写字母	集合 { a b c d e f g h i j k l m n o p q r s t u v w x y z }
大写字母	集合 {A B C D E F G H I J K L M N O P Q R S T U V W X Y Z }
字母	小写字母和大写字母的集合
字母数字字符	数字、小写字母和大写字母的集合
标点符号字符	集合 ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~
图形字符	字母数字字符和标点符号字符的集合
空格字符	制表符、换行符、垂直制表符、换页符、回车符、空格符的集合。
可打印字符	字母数字字符、标点符号字符和空格字符的集合。
控制字符	在 ASCII 编码中，这些字符的八进制代码是从 000 到 037，以及 177（DEL）。
空白字符	包括空格符和制表符。
字母字符	小写字母和大写字母的集合。

C 库函数 - isalnum()

描述

C 库函数 **void isalnum(int c)** 检查所传的字符是否是字母和数字。

声明

下面是 isalnum() 函数的声明。

```
int isalnum(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个数字或一个字母，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isalnum() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'd';
   int var2 = '2';
   int var3 = '\t';
   int var4 = ' ';

   if( isalnum(var1) )
   {
      printf("var1 = |%c| 是字母数字\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是字母数字\n", var1 );
   }
   if( isalnum(var2) )
   {
      printf("var2 = |%c| 是字母数字\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是字母数字\n", var2 );
   }
   if( isalnum(var3) )
   {
      printf("var3 = |%c| 是字母数字\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是字母数字\n", var3 );
   }
   if( isalnum(var4) )
   {
      printf("var4 = |%c| 是字母数字\n", var4 );
   }
   else
   {
      printf("var4 = |%c| 不是字母数字\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |d| 是字母数字
var2 = |2| 是字母数字
var3 = |	| 不是字母数字
var4 = | | 不是字母数字
```

C 库函数 - isalpha()

描述

C 库函数 **void isalpha(int c)** 检查所传的字符是否是字母。

声明

下面是 isalpha() 函数的声明。

```
int isalpha(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个字母，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isalpha() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'd';
   int var2 = '2';
   int var3 = '\t';
   int var4 = ' ';

   if( isalpha(var1) )
   {
      printf("var1 = |%c| 是一个字母\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是一个字母\n", var1 );
   }
   if( isalpha(var2) )
   {
      printf("var2 = |%c| 是一个字母\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是一个字母\n", var2 );
   }
   if( isalpha(var3) )
   {
      printf("var3 = |%c| 是一个字母\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是一个字母\n", var3 );
   }
   if( isalpha(var4) )
   {
      printf("var4 = |%c| 是一个字母\n", var4 );
   }
   else
   {
      printf("var4 = |%c| 不是一个字母\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |d| 是一个字母
var2 = |2| 不是一个字母
var3 = | | 不是一个字母
var4 = | | 不是一个字母
```

C 库函数 - iscntrl()

描述

C 库函数 **void iscntrl(int c)** 检查所传的字符是否是控制字符。

根据标准 ASCII 字符集，控制字符的 ASCII 编码介于 0x00 (NUL) 和 0x1f (US) 之间，以及 0x7f (DEL)，某些平台的特定编译器实现还可以在扩展字符集（0x7f 以上）中定义额外的控制字符。

声明

下面是 iscntrl() 函数的声明。

```
int iscntrl(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个控制字符，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 iscntrl() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main ()
{
   int i = 0, j = 0;
   char str1[] = "all \a about \t programming";
   char str2[] = "w3cschool \n tutorials";

   /* 输出字符串直到控制字符 \a */
   while( !iscntrl(str1[i]) )
   {
      putchar(str1[i]);
      i++;
   }

   /* 输出字符串直到控制字符 \n */
   while( !iscntrl(str2[j]) )
   {
      putchar(str2[j]);
      j++;
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
all w3cschool
```

C 库函数 - isdigit()

描述

C 库函数 **void isdigit(int c)** 检查所传的字符是否是十进制数字字符。

十进制数字是：0 1 2 3 4 5 6 7 8 9

声明

下面是 isdigit() 函数的声明。

```
int isdigit(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个数字，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isdigit() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'h';
   int var2 = '2';

   if( isdigit(var1) )
   {
      printf("var1 = |%c| 是一个数字\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是一个数字\n", var1 );
   }
   if( isdigit(var2) )
   {
      printf("var2 = |%c| 是一个数字\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是一个数字\n", var2 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |h| 不是一个数字
var2 = |2| 是一个数字
```

C 库函数 - isgraph()

描述

C 库函数 **void isgraph(int c)** 检查所传的字符是否有图形表示法。

带有图形表示法的字符是除了空白字符（比如 ' '）以外的所有可打印的字符。

声明

下面是 isgraph() 函数的声明。

```
int isgraph(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 有图形表示法，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isgraph() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = '3';
   int var2 = 'm';
   int var3 = ' ';

   if( isgraph(var1) )
   {
       printf("var1 = |%c| 是可打印的\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 是不可打印的\n", var1 );
   }
   if( isgraph(var2) )
   {
       printf("var2 = |%c| 是可打印的\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 是不可打印的\n", var2 );
   }
   if( isgraph(var3) )
   {
       printf("var3 = |%c| 是可打印的\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 是不可打印的\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |3| 是可打印的
var2 = |m| 是可打印的
var3 = | | 是不可打印的
```

C 库函数 - islower()

描述

C 库函数 **int islower(int c)** 检查所传的字符是否是小写字母。

声明

下面是 islower() 函数的声明。

```
int islower(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个小写字母，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 islower() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'Q';
   int var2 = 'q';
   int var3 = '3';

   if( islower(var1) )
   {
       printf("var1 = |%c| 是小写字母\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是小写字母\n", var1 );
   }
   if( islower(var2) )
   {
       printf("var2 = |%c| 是小写字母\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是小写字母\n", var2 );
   }
   if( islower(var3) )
   {
       printf("var3 = |%c| 是小写字母\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是小写字母\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |Q| 不是小写字母
var2 = |q| 是小写字母
var3 = |3| 不是小写字母
```

C 库函数 - isprint()

描述

C 库函数 **int isprint(int c)** 检查所传的字符是否是可打印的。可打印字符是非控制字符的字符。

声明

下面是 isprint() 函数的声明。

```
int isprint(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个可打印的字符，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isprint() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'k';
   int var2 = '8';
   int var3 = '\t';
   int var4 = ' ';

   if( isprint(var1) )
   {
      printf("var1 = |%c| 是可打印的\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 是不可打印的\n", var1 );
   }
   if( isprint(var2) )
   {
      printf("var2 = |%c| 是可打印的\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 是不可打印的\n", var2 );
   }
   if( isprint(var3) )
   {
      printf("var3 = |%c| 是可打印的\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 是不可打印的\n", var3 );
   }
   if( isprint(var4) )
   {
      printf("var4 = |%c| 是可打印的\n", var4 );
   }
   else
   {
      printf("var4 = |%c| 是不可打印的\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |k| 是可打印的
var2 = |8| 是可打印的
var3 = |	| 是不可打印的
var4 = | | 是可打印的
```

C 库函数 - ispunct()

描述

C 库函数 **int ispunct(int c)** 检查所传的字符是否是标点符号字符。标点符号字符可以是非字母数字（正如 isalnum 中的一样）的任意图形字符（正如 isgraph 中的一样）。

声明

下面是 ispunct() 函数的声明。

```
int ispunct(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个标点符号字符，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 ispunct() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 't';
   int var2 = '1';
   int var3 = '/';
   int var4 = ' ';

   if( ispunct(var1) )
   {
       printf("var1 = |%c| 是标点符号字符\n", var1 );
   }
   else
   {
       printf("var1 = |%c| 不是标点符号字符\n", var1 );
   }
   if( ispunct(var2) )
   {
       printf("var2 = |%c| 是标点符号字符\n", var2 );
   }
   else
   {
       printf("var2 = |%c| 不是标点符号字符\n", var2 );
   }
   if( ispunct(var3) )
   {
       printf("var3 = |%c| 是标点符号字符\n", var3 );
   }
   else
   {
       printf("var3 = |%c| 不是标点符号字符\n", var3 );
   }
   if( ispunct(var4) )
   {
       printf("var4 = |%c| 是标点符号字符\n", var4 );
   }
   else
   {
       printf("var4 = |%c| 不是标点符号字符\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |t| 不是标点符号字符
var2 = |1| 不是标点符号字符
var3 = |/| 是标点符号字符
var4 = | | 不是标点符号字符
```

C 库函数 - isspace()

描述

C 库函数 **int isspace(int c)** 检查所传的字符是否是空白字符。

标准的空白字符包括：

```
' '     (0x20)    space (SPC) 空格符
'\t'    (0x09)    horizontal tab (TAB) 水平制表符
'\n'    (0x0a)    newline (LF) 换行符
'\v'    (0x0b)    vertical tab (VT) 垂直制表符
'\f'    (0x0c)    feed (FF) 换页符
'\r'    (0x0d)    carriage return (CR) 回车符
```

声明

下面是 isspace() 函数的声明。

```
int isspace(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个空白字符，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isspace() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 't';
   int var2 = '1';
   int var3 = ' ';

   if( isspace(var1) )
   {
       printf("var1 = |%c| 是空白字符\n", var1 );
   }
   else
   {
       printf("var1 = |%c| 不是空白字符\n", var1 );
   }
   if( isspace(var2) )
   {
       printf("var2 = |%c| 是空白字符\n", var2 );
   }
   else
   {
       printf("var2 = |%c| 不是空白字符\n", var2 );
   }
   if( isspace(var3) )
   {
       printf("var3 = |%c| 是空白字符\n", var3 );
   }
   else
   {
       printf("var3 = |%c| 不是空白字符\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |t| 不是空白字符
var2 = |1| 不是空白字符
var3 = | | 是空白字符
```

C 库函数 - isupper()

描述

C 库函数 **int isupper(int c)** 检查所传的字符是否是大写字母。

声明

下面是 isupper() 函数的声明。

```
int isupper(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个大写字母，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isupper() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'M';
   int var2 = 'm';
   int var3 = '3';

   if( isupper(var1) )
   {
      printf("var1 = |%c| 是大写字母\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是大写字母\n", var1 );
   }
   if( isupper(var2) )
   {
      printf("var2 = |%c| 是大写字母\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是大写字母\n", var2 );
   }
   if( isupper(var3) )
   {
      printf("var3 = |%c| 是大写字母\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是大写字母\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |M| 是大写字母
var2 = |m| 不是大写字母
var3 = |3| 不是大写字母
```

C 库函数 - isxdigit()

描述

C 库函数 **int isxdigit(int c)**检查所传的字符是否是十六进制数字。

声明

下面是 isxdigit() 函数的声明。

```
int isxdigit(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个十六进制数字，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isxdigit() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   char var1[] = "tuts";
   char var2[] = "0xE";

   if( isxdigit(var1[0]) )
   {
     printf("var1 = |%s| 是十六进制数字\n", var1 );
   }
   else
   {
     printf("var1 = |%s| 不是十六进制数字\n", var1 );
   }

   if( isxdigit(var2[0] ))
   {
     printf("var2 = |%s| 是十六进制数字\n", var2 );
   }
   else
   {
     printf("var2 = |%s| 不是十六进制数字\n", var2 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |tuts| 不是十六进制数字
var2 = |0xE| 是十六进制数字
```

C 标准库 - <errno.h>

简介

C 标准库的 **errno.h** 头文件定义了整数变量 **errno**，它是通过系统调用设置的，在错误事件中的某些库函数表明了什么发生了错误。该宏扩展为类型为 int 的可更改的左值，因此它可以被一个程序读取和修改。

在程序启动时，**errno** 设置为零，C 标准库中的特定函数修改它的值为一些非零值以表示某些类型的错误。您也可以在适当的时候修改它的值或重置为零。

errno.h 头文件也顶了以一系列表示不同错误代码的宏，这些宏应扩展为类型为 **int** 的整数常量表达式。

库宏

下面列出了头文件 errno.h 中定义的宏：

宏	描述
extern int errno	这是通过系统调用设置的宏，在错误事件中的某些库函数表明了什么发生了错误。
EDOM Domain Error	这个宏表示一个域错误，它在输入参数超出数学函数定义的域时发生，errno 被设置为 EDOM。
ERANGE Range Error	这个宏表示一个范围错误，它在输入参数超出数学函数定义的范围时发生，errno 被设置为 ERANGE。

C 库宏 - errno

描述

C 库宏 **extern int errno** 是通过系统调用设置的，在错误事件中的某些库函数表明了什么发生了错误。

声明

下面是 errno 宏的声明。

```
extern int errno
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 errno 宏的用法。

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      fprintf(stderr, "Value of errno: %d\n", errno);
      fprintf(stderr, "Error opening file: %s\n", strerror(errno));
   }
   else
   {
      fclose(fp);
   }

   return(0);
}
```

让我们编译并运行上面的程序，当文件 **file.txt** 不存在时，将产生以下结果：

```
Value of errno: 2
Error opening file: No such file or directory
```

C 库宏 - EDOM

描述

C 库宏 **EDOM** 表示一个域错误，它在输入参数超出数学函数定义的域时发生，errno 被设置为 EDOM。

声明

下面是 EDOM 宏的声明。

```
#define EDOM some_value
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 EDOM 宏的用法。

```
#include <stdio.h>
#include <errno.h>
#include <math.h>

int main()
{
   double val;

   errno = 0;
   val = sqrt(-10);
   if(errno == EDOM)
   {
      printf("Invalid value \n");
   }
   else
   {
      printf("Valid value\n");
   }

   errno = 0;
   val = sqrt(10);
   if(errno == EDOM)
   {
      printf("Invalid value\n");
   }
   else
   {
      printf("Valid value\n");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Invalid value
Valid value
```

C 库宏 - ERANGE

描述

C 库宏 **ERANGE** 表示一个范围错误，它在输入参数超出数学函数定义的范围时发生，errno 被设置为 ERANGE。

声明

下面是 ERANGE 宏的声明。

```
#define ERANGE some_value
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 ERANGE 宏的用法。

```
#include <stdio.h>
#include <errno.h>
#include <math.h>

int main()
{
   double x;
   double value;

   x = 1.000000;
   value = log(x);
   if( errno == ERANGE )
   {
      printf("Log(%f) is out of range\n", x);
   }
   else
   {
      printf("Log(%f) = %f\n", x, value);
   }

   x = 0.000000;
   value = log(x);
   if( errno == ERANGE )
   {
      printf("Log(%f) is out of range\n" x);
   }
   else
   {
      printf("Log(%f) = %f\n", x, value);
   }

   return 0;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Log(1.000000) = 1.609438
Log(0.000000) is out of range
```

C 标准库 - <float.h>

简介

C 标准库的 **float.h** 头文件包含了一组与浮点值相关的依赖于平台的常量。这些常量是由 ANSI C 提出的，这让程序更具有可移植性。在讲解这些常量之前，最好先弄清楚浮点数是由下面四个元素组成的：

组件	组件描述
S	符号 (+/-)
b	指数表示的基数，2 表示二进制，10 表示十进制，16 表示十六进制，等等...
e	指数，一个介于最小值 e_{min} 和最大值 e_{max} 之间的整数。
p	精度，基数 b 的有效位数

基于以上 4 个组成部分，一个浮点数的值如下：

```
floating-point = ( S ) p x b<sup>e</sup>

或

floating-point = (+/-) precision x base<sup>exponent</sup>
```

库宏

下面的值是特定实现的，且是通过 #define 指令来定义的，这些值都不得低于下边所给出的值。请注意，所有的实例 FLT 是指类型 float，DBL 是指类型 double，LDBL 是指类型 long double。

宏	描述
FLT_ROUNDS	定义浮点加法的舍入模式，它可以是下列任何一个值：-1 - 无法确定 0 - 趋向于零 1 - 去最近的值 2 - 趋向于正无穷 3 - 趋向于负无穷
FLT_RADIX 2	这个宏定义了指数表示的基数。基数 2 表示二进制，基数 10 表示十进制，基数 16 表示十六进制。
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	这些宏定义了 FLT_RADIX 基数中的位数。
FLT_DIG 6 DBL_DIG 10 LDBL_DIG 10	这些宏定义了舍入后不会改变表示的十进制数字的最大值（基数 10）。
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	这些宏定义了基数为 FLT_RADIX 时的指数的最小负整数值。
FLT_MIN_10_EXP -37 DBL_MIN_10_EXP -37 LDBL_MIN_10_EXP -37	这些宏定义了基数为 10 时的指数的最小负整数值。
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	这些宏定义了基数为 FLT_RADIX 时的指数的最大整数值。
FLT_MAX_10_EXP +37 DBL_MAX_10_EXP +37 LDBL_MAX_10_EXP +37	这些宏定义了基数为 10 时的指数的最大整数值。
FLT_MAX 1E+37 DBL_MAX 1E+37 LDBL_MAX 1E+37	这些宏定义最大的有限浮点值。
FLT_EPSILON 1E-5 DBL_EPSILON 1E-9 LDBL_EPSILON 1E-9	这些宏定义了可表示的最小有效数字。
FLT_MIN 1E-37 DBL_MIN 1E-37 LDBL_MIN 1E-37	这些宏定义了最小的浮点值。

实例

下面的实例演示了 float.h 文件中定义的一些常量的使用。

```
#include <stdio.h>
#include <float.h>

int main()
{
   printf("The maximum value of float = %.10e\n", FLT_MAX);
   printf("The minimum value of float = %.10e\n", FLT_MIN);

   printf("The number of digits in the number = %.10e\n", FLT_MANT_DIG);
}
```

让我们编译和运行上面的程序，这将产生下列结果：

```
The maximum value of float = 3.4028234664e+38
The minimum value of float = 1.1754943508e-38
The number of digits in the number = 7.2996655210e-312
```

C 标准库 - <limits.h>

简介

limits.h 头文件决定了各种变量类型的各种属性。定义在该头文件中的宏限制了各种变量类型（比如 char、int 和 long）的值。

这些限制指定了变量不能存储任何超出这些限制的值，例如一个无符号可以存储的最大值是 255。

库宏

下面的值是特定实现的，且是通过 #define 指令来定义的，这些值都不得低于下边所给出的值。

宏	值	描述
CHAR_BIT	8	定义一个字节的比特数。
SCHAR_MIN	-128	定义一个有符号字符的最小值。
SCHAR_MAX	127	定义一个有符号字符的最大值。
UCHAR_MAX	255	定义一个无符号字符的最大值。
CHAR_MIN	0	定义类型 char 的最小值，如果 char 表示负值，则它的值等于 SCHAR_MIN，否则等于 0。
CHAR_MAX	127	定义类型 char 的最大值，如果 char 表示负值，则它的值等于 SCHAR_MAX，否则等于 UCHAR_MAX。
MB_LEN_MAX	1	定义多字节字符中的最大字节数。
SHRT_MIN	-32768	定义一个短整型的最小值。
SHRT_MAX	+32767	定义一个短整型的最大值。
USHRT_MAX	65535	定义一个无符号短整型的最大值。
INT_MIN	-32768	定义一个整型的最小值。
INT_MAX	+32767	定义一个整型的最大值。
UINT_MAX	65535	定义一个无符号整型的最大值。
LONG_MIN	-2147483648	定义一个长整型的最小值。
LONG_MAX	+2147483647	定义一个长整型的最大值。
ULONG_MAX	4294967295	定义一个无符号长整型的最大值。

实例

下面的实例演示了 limit.h 文件中定义的一些常量的使用。

```
#include <stdio.h>
#include <limits.h>

int main()
{

   printf("The number of bits in a byte %d\n", CHAR_BIT);

   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);

   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);

   return(0);
}
```

让我们编译和运行上面的程序，这将产生下列结果：

```
The number of bits in a byte 8
The minimum value of SIGNED CHAR = -128
The maximum value of SIGNED CHAR = 127
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -32768
The maximum value of INT = 32767
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -2147483648
The maximum value of LONG = 2147483647
```

C 标准库 - <locale.h>

简介

locale.h 头文件定义了特定地域的设置，比如日期格式和货币符号。接下来我们将介绍一些宏，以及一个重要的结构 **struct lconv** 和两个重要的函数。

库宏

下面列出了头文件 locale.h 中定义的宏，这些宏将在下列的两个函数中使用：

宏	描述
LC_ALL	设置下面的所有选项。
LC_COLLATE	影响 strcoll 和 strxfrm 函数。
LC_CTYPE	影响所有字符函数。
LC_MONETARY	影响 localeconv 函数提供的货币信息。
LC_NUMERIC	影响 localeconv 函数提供的小数点格式化和信息。
LC_TIME	影响 strftime 函数。

库函数

下面列出了头文件 locale.h 中定义的函数：

函数	描述
char *setlocale(int category, const char *locale)	设置或读取地域化信息。
struct lconv *localeconv(void)	设置或读取地域化信息。

库结构

```
typedef struct {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_space;
   char n_cs_precedes;
   char n_sep_by_space;
   char p_sign_posn;
   char n_sign_posn;
} lconv
```

以下是各字段的描述：

字段	描述
decimal_point	用于非货币值的小数点字符。
thousands_sep	用于非货币值的千位分隔符。
grouping	一个表示非货币量中每组数字大小的字符串。每个字符代表一个整数值，每个整数指定当前组的位数。值为 0 意味着前一个值将应用于剩余的分组。
int_curr_symbol	国际货币符号使用的字符串。前三个字符是由 ISO 4217:1987 指定的，第四个字符用于分隔货币符号和货币量。
currency_symbol	用于货币的本地符号。
mon_decimal_point	用于货币值的小数点字符。
mon_thousands_sep	用于货币值的千位分隔符。
mon_grouping	一个表示货币值中每组数字大小的字符串。每个字符代表一个整数值，每个整数指定当前组的位数。值为 0 意味着前一个值将应用于剩余的分组。
positive_sign	用于正货币值的字符。
negative_sign	用于负货币值的字符。
int_frac_digits	国际货币值中小数点后要显示的位数。
frac_digits	货币值中小数点后要显示的位数。
p_cs_precedes	如果等于 1，则 currency_symbol 出现在正货币值之前。如果等于 0，则 currency_symbol 出现在正货币值之后。
p_sep_by_space	如果等于 1，则 currency_symbol 和正货币值之间使用空格分隔。如果等于 0，则 currency_symbol 和正货币值之间不使用空格分隔。
n_cs_precedes	如果等于 1，则 currency_symbol 出现在负货币值之前。如果等于 0，则 currency_symbol 出现在负货币值之后。
n_sep_by_space	如果等于 1，则 currency_symbol 和负货币值之间使用空格分隔。如果等于 0，则 currency_symbol 和负货币值之间不使用空格分隔。
p_sign_posn	表示正货币值中正号的位置。
n_sign_posn	表示负货币值中负号的位置。

下面的值用于 **p_sign_posn** 和 **n_sign_posn**:

值	描述
0	封装值和 currency_symbol 的括号。
1	放置在值和 currency_symbol 之前的符号。
2	放置在值和 currency_symbol 之后的符号。
3	紧挨着放置在值和 currency_symbol 之前的符号。
4	紧挨着放置在值和 currency_symbol 之后的符号。

C 库函数 - setlocale()

描述

C 库函数 **char *setlocale(int category, const char* locale)** 设置或读取地域化信息。

声明

下面是 setlocale() 函数的声明。

```
char *setlocale(int category, const char *locale)
```

参数

- **category** -- 这是一个已命名的常量，指定了受区域设置影响的函数类别。
  - **LC_ALL** 包括下面的所有选项。
  - **LC_COLLATE** 字符串比较。参见 strcoll()。
  - **LC_CTYPE** 字符分类和转换。例如 strtoupper()。
  - **LC_MONETARY** 货币格式，针对 localeconv()。
  - **LC_NUMERIC** 小数点分隔符，针对 localeconv()。
  - **LC_TIME** 日期和时间格式，针对 strftime()。
  - **LC_MESSAGES** 系统响应。
- **locale** -- 如果 locale 是 NULL 或空字符串 ""，则区域名称将根据环境变量值来设置，其名称与上述的类别名称相同。

返回值

如果成功调用 setlocale()，则返回一个对应于区域设置的不透明的字符串。如果请求无效，则返回值是 NULL。

实例

下面的实例演示了 setlocale() 函数的用法。

```
#include <locale.h>
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t currtime;
   struct tm *timer;
   char buffer[80];

   time( &currtime );
   timer = localtime( &currtime );

   printf("Locale is: %s\n", setlocale(LC_ALL, "en_GB"));
   strftime(buffer,80,"%c", timer );
   printf("Date is: %s\n", buffer);

   printf("Locale is: %s\n", setlocale(LC_ALL, "de_DE"));
   strftime(buffer,80,"%c", timer );
   printf("Date is: %s\n", buffer);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Locale is: en_GB
Date is: Thu 23 Aug 2012 06:39:32 MST
Locale is: de_DE
Date is: Do 23 Aug 2012 06:39:32 MST
```

C 库函数 - localeconv()

描述

C 库函数 **struct lconv *localeconv(void)** 设置或读取地域化信息。它会返回一个 **lconv** 结构类型的对象。

声明

下面是 localeconv() 函数的声明。

```
struct lconv *localeconv(void)
```

参数

- **NA**

返回值

该函数返回一个指向当前区域 **struct lconv** 的指针，它的结构如下：

```
typedef struct {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_space;
   char n_cs_precedes;
   char n_sep_by_space;
   char p_sign_posn;
   char n_sign_posn;
} lconv
```

实例

下面的实例演示了 localeconv() 函数的用法。

```
#include <locale.h>
#include <stdio.h>

int main ()
{
   struct lconv * lc;

   setlocale(LC_MONETARY, "it_IT");
   lc = localeconv();
   printf("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf("International Currency Symbol: %s\n",lc->int_curr_symbol);

   setlocale(LC_MONETARY, "en_US");
   lc = localeconv();
   printf("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf("International Currency Symbol: %s\n",lc->int_curr_symbol);

   setlocale(LC_MONETARY, "en_GB");
   lc = localeconv();
   printf ("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf ("International Currency Symbol: %s\n",lc->int_curr_symbol);

   printf("Decimal Point = %s\n", lc->decimal_point);

   return 0;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Local Currency Symbol: EUR
International Currency Symbol: EUR
Local Currency Symbol: $
International Currency Symbol: USD
Local Currency Symbol: £
International Currency Symbol: GBP
Decimal Point = .
```

C 标准库 - <math.h>

简介

math.h 头文件定义了各种数学函数和一个宏。在这个库中所有可用的功能都带有一个 **double** 类型的参数，且都返回 **double** 类型的结果。

库宏

下面是这个库中定义的唯一的一个宏：

宏	描述
HUGE_VAL	当函数的结果不可以表示为浮点数时。如果是因为结果的幅度太大以致于无法表示，则函数会设置 errno 为 ERANGE 来表示范围错误，并返回一个由宏 HUGE_VAL 或者它的否定（- HUGE_VAL）命名的一个特定的很大的值。 如果结果的幅度太小，则会返回零值。在这种情况下，error 可能会被设置为 ERANGE，也有可能不会被设置为 ERANGE。

库函数

下面列出了头文件 math.h 中定义的函数：

函数	描述
double acos(double x)	返回以弧度表示的 x 的反余弦。
double asin(double x)	返回以弧度表示的 x 的反正弦。
double atan(double x)	返回以弧度表示的 x 的反正切。
double atan2(double y, double x)	返回以弧度表示的 y/x 的反正切。y 和 x 的值的符号决定了正确的象限。
double cos(double x)	返回弧度角 x 的余弦。
double cosh(double x)	返回 x 的双曲余弦。
double sin(double x)	返回弧度角 x 的正弦。

double sinh(double x)	返回 x 的双曲正弦。
double tanh(double x)	返回 x 的双曲正切。
double exp(double x)	返回 e 的 x 次幂的值。
double frexp(double x, int *exponent)	把浮点数 x 分解成尾数和指数。返回值是尾数，并将指数存入 exponent 中。所得的值是 x = mantissa * 2 ^ exponent。
double ldexp(double x, int exponent)	返回 x 乘以 2 的 exponent 次幂。
double log(double x)	返回 x 的自然对数（基数为 e 的对数）。
double log10(double x)	返回 x 的常用对数（基数为 10 的对数）。
double modf(double x, double *integer)	返回值为小数部分（小数点后的部分），并设置 integer 为整数部分。
double pow(double x, double y)	返回 x 的 y 次幂。
double sqrt(double x)	返回 x 的平方根。
double ceil(double x)	返回大于或等于 x 的最小的整数值。
double fabs(double x)	返回 x 的绝对值。
double floor(double x)	返回小于或等于 x 的最大的整数值。
double fmod(double x, double y)	返回 x 除以 y 的余数。

C 库函数 - acos()

描述

C 库函数 **double acos(double x)** 返回以弧度表示的 **x** 的反余弦。

声明

下面是 acos() 函数的声明。

```
double acos(double x)
```

参数

- **x** -- 介于 [-1,+1] 区间的浮点值。

返回值

该函数返回以弧度表示的 x 的反余弦，弧度区间为 [0, pi]。

实例

下面的实例演示了 acos() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 0.9;
   val = 180.0 / PI;

   ret = acos(x) * val;
   printf("%lf 的反余弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.900000 的反余弦是 25.855040 度
```

C 库函数 - asin()

描述

C 库函数 **double asin(double x)** 返回以弧度表示的 **x** 的反正弦。

声明

下面是 asin() 函数的声明。

```
double asin(double x)
```

参数

- **x** -- 介于 [-1,+1] 区间的浮点值。

返回值

该函数返回以弧度表示的 x 的反正弦，弧度区间为 [-pi/2,+pi/2]。

实例

下面的实例演示了 asin() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;
   x = 0.9;
   val = 180.0 / PI;

   ret = asin(x) * val;
   printf("%lf 的反正弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.900000 的反正弦是 64.190609 度
```

C 库函数 - atan()

描述

C 库函数 **double atan(double x)** 返回以弧度表示的 **x** 的反正切。

声明

下面是 atan() 函数的声明。

```
double atan(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回以弧度表示的 x 的反正切，弧度区间为 [-pi/2,+pi/2]。

实例

下面的实例演示了 atan() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;
   x = 1.0;
   val = 180.0 / PI;

   ret = atan (x) * val;
   printf("%lf 的反正切是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
1.000000 的反正切是 45.000000 度
```

C 库函数 - atan2()

描述

C 库函数 **double atan2(doubly y, double x)** 返回以弧度表示的 **y/x** 的反正切。y 和 x 的值的符号决定了正确的象限。

声明

下面是 atan2() 函数的声明。

```
double atan2(doubly y, double x)
```

参数

- **x** -- 代表 x 轴坐标的浮点值。
- **y** -- 代表 y 轴坐标的浮点值。

返回值

该函数返回以弧度表示的 y/x 的反正切，弧度区间为 [-pi,+pi]。

实例

下面的实例演示了 atan2() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, y, ret, val;

   x = -7.0;
   y = 7.0;
   val = 180.0 / PI;

   ret = atan2 (y,x) * val;
   printf("x = %lf, y = %lf 的反正切", x, y);
   printf("是 %lf 度\n", ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
x = -7.000000, y = 7.000000 的反正切是 135.000000 度
```

C 库函数 - cos()

描述

C 库函数 **double cos(double x)** 返回弧度角 **x** 的余弦。

声明

下面是 cos() 函数的声明。

```
double cos(double x)
```

参数

- **x** -- 浮点值，代表了一个以弧度表示的角度。

返回值

该函数返回 x 的余弦。

实例

下面的实例演示了 cos() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 60.0;
   val = PI / 180.0;
   ret = cos( x*val );
   printf("%lf 的余弦是 %lf 度\n", x, ret);

   x = 90.0;
   val = PI / 180.0;
   ret = cos( x*val );
   printf("%lf 的余弦是 %lf 度\n", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
60.000000 的余弦是 0.500000 度
90.000000 的余弦是 0.000000 度
```

C 库函数 - cosh()

描述

C 库函数 **double cosh(double x)** 返回 **x** 的双曲余弦。

声明

下面是 cosh() 函数的声明。

```
double cosh(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的双曲余弦。

实例

下面的实例演示了 cosh() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x;

   x = 0.5;
   printf("%lf 的双曲余弦是 %lf\n", x, cosh(x));

   x = 1.0;
   printf("%lf 的双曲余弦是 %lf\n", x, cosh(x));

   x = 1.5;
   printf("%lf 的双曲余弦是 %lf\n", x, cosh(x));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.500000 的双曲余弦是 1.127626
1.000000 的双曲余弦是 1.543081
1.500000 的双曲余弦是 2.352410
```

C 库函数 - sin()

描述

C 库函数 **double sin(double x)** 返回弧度角 **x** 的正弦。

声明

下面是 sin() 函数的声明。

```
double sin(double x)
```

参数

- **x** -- 浮点值，代表了一个以弧度表示的角度。

返回值

该函数返回 x 的正弦。

实例

下面的实例演示了 sin() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 45.0;
   val = PI / 180;
   ret = sin(x*val);
   printf("%lf 的正弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
45.000000 的正弦是 0.707107 度
```

C 库函数 - sinh()

描述

C 库函数 **double sinh(double x)** 返回 **x** 的双曲正弦。

声明

下面是 sinh() 函数的声明。

```
double sinh(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的双曲正弦。

实例

下面的实例演示了 sinh() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 0.5;

   ret = sinh(x);
   printf("%lf 的双曲正弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.500000 的双曲正弦是 0.521095 度
```

C 库函数 - tanh()

描述

C 库函数 **double tanh(double x)** 返回 **x** 的双曲正切。

声明

下面是 tanh() 函数的声明。

```
double tanh(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的双曲正切。

实例

下面的实例演示了 tanh() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 0.5;

   ret = tanh(x);
   printf("%lf 的双曲正切是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.500000 的双曲正切是 0.462117 度
```

C 库函数 - exp()

描述

C 库函数 **double exp(double x)** 返回 **e** 的 **x** 次幂的值。

声明

下面是 exp() 函数的声明。

```
double exp(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 e 的 x 次幂。

实例

下面的实例演示了 exp() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x = 0;

   printf("e 的 %lf 次幂是 %lf\n", x, exp(x));
   printf("e 的 %lf 次幂是 %lf\n", x+1, exp(x+1));
   printf("e 的 %lf 次幂是 %lf\n", x+2, exp(x+2));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
e 的 0.000000 次幂是 1.000000
e 的 1.000000 次幂是 2.718282
e 的 2.000000 次幂是 7.389056
```

C 库函数 - frexp()

描述

C 库函数 **double frexp(double x, int *exponent)** 把浮点数 x 分解成尾数和指数。返回值是尾数，并将指数存入 **exponent** 中。所得的值是 **x = mantissa * 2 ^ exponent**。

声明

下面是 frexp() 函数的声明。

```
double frexp(double x, int *exponent)
```

参数

- **x** -- 要被计算的浮点值。
- **exponent** -- 指向一个对象的指针，该对象存储了指数的值。

返回值

该函数返回规格化小数。如果参数 x 不为零，则规格化小数是 x 的二次方，且它的绝对值范围从 1/2（包含）到 1（不包含）。如果 x 为零，则规格化小数是零，且零存储在 exp 中。

实例

下面的实例演示了 frexp() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x = 1024, fraction;
   int e;

   fraction = frexp(x, &e);
   printf("x = %.2lf = %.2lf * 2^%d\n", x, fraction, e);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
x = 1024.00 = 0.50 * 2^11
```

C 库函数 - ldexp()

描述

C 库函数 **double ldexp(double x, int exponent)** 返回 **x** 乘以 2 的 **exponent** 次幂。

声明

下面是 ldexp() 函数的声明。

```
double ldexp(double x, int exponent)
```

参数

- **x** -- 代表有效位数的浮点值。
- **exponent** -- 指数的值。

返回值

该函数返回 x * 2 ^{exp}。

实例

下面的实例演示了 ldexp() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   int n;

   x = 0.65;
   n = 3;
   ret = ldexp(x ,n);
   printf("%f * 2^%d = %f\n", x, n, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.650000 * 2^3 = 5.200000
```

C 库函数 - log()

描述

C 库函数 **double log(double x)** 返回 **x** 的自然对数（基数为 e 的对数）。

声明

下面是 log() 函数的声明。

```
double log(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的自然对数。

实例

下面的实例演示了 log() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 2.7;

   /* 计算 log(2.7) */
   ret = log(x);
   printf("log(%lf) = %lf", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
log(2.700000) = 0.993252
```

C 库函数 - log10()

描述

C 库函数 **double log10(double x)** 返回 **x** 的常用对数（基数为 10 的对数）。

声明

下面是 log10() 函数的声明。

```
double log10(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的常用对数，x 的值大于 0。

实例

下面的实例演示了 log10() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 10000;

   /* 计算 log10(10000) */
   ret = log10(x);
   printf("log10(%lf) = %lf\n", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
log10(10000.000000) = 4.000000
```

C 库函数 - modf()

描述

C 库函数 **double modf(double x, double *integer)** 返回值为小数部分（小数点后的部分），并设置 integer 为整数部分。

声明

下面是 modf() 函数的声明。

```
double modf(double x, double *integer)
```

参数

- **x** -- 浮点值。
- **integer** -- 指向一个对象的指针，该对象存储了整数部分。

返回值

该函数返回 x 的小数部分，符号与 x 相同。

实例

下面的实例演示了 modf() 函数的用法。

```
#include<stdio.h>
#include<math.h>

int main ()
{
   double x, fractpart, intpart;

   x = 8.123456;
   fractpart = modf(x, &intpart);

   printf("整数部分 = %lf\n", intpart);
   printf("小数部分 = %lf \n", fractpart);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
整数部分 = 8.000000
小数部分 = 0.123456
```

C 库函数 - pow()

描述

C 库函数 **double pow(double x, double y)** 返回 **x** 的 **y** 次幂，即 x^y。

声明

下面是 pow() 函数的声明。

```
double pow(double x, double y)
```

参数

- **x** -- 代表基数的浮点值。
- **y** -- 代表指数的浮点值。

返回值

该函数返回 x 的 y 次幂的结果。

实例

下面的实例演示了 pow() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   printf("值 8.0 ^ 3 = %lf\n", pow(8.0, 3));

   printf("值 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
值 8.0 ^ 3 = 512.000000
值 3.05 ^ 1.98 = 9.097324
```

C 库函数 - sqrt()

描述

C 库函数 **double sqrt(double x)** 返回 **x** 的平方根。

声明

下面是 sqrt() 函数的声明。

```
double sqrt(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的平方根。

实例

下面的实例演示了 sqrt() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{

   printf("%lf 的平方根是 %lf\n", 4.0, sqrt(4.0) );
   printf("%lf 的平方根是 %lf\n", 5.0, sqrt(5.0) );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
4.000000 的平方根是 2.000000
5.000000 的平方根是 2.236068
```

C 库函数 - ceil()

描述

C 库函数 **double ceil(double x)** 返回大于或等于 **x** 的最小的整数值。

声明

下面是 ceil() 函数的声明。

```
double ceil(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回不小于 x 的最小整数值。

实例

下面的实例演示了 ceil() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   float val1, val2, val3, val4;

   val1 = 1.6;
   val2 = 1.2;
   val3 = 2.8;
   val4 = 2.3;

   printf ("value1 = %.1lf\n", ceil(val1));
   printf ("value2 = %.1lf\n", ceil(val2));
   printf ("value3 = %.1lf\n", ceil(val3));
   printf ("value4 = %.1lf\n", ceil(val4));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
value1 = 2.0
value2 = 2.0
value3 = 3.0
value4 = 3.0
```

C 库函数 - fabs()

描述

C 库函数 **double fabs(double x)** 返回 **x** 的绝对值。

声明

下面是 fabs() 函数的声明。

```
double fabs(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的绝对值。

实例

下面的实例演示了 fabs() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   int a, b;
   a = 1234;
   b = -344;

   printf("%d 的绝对值是 %lf\n", a, fabs(a));
   printf("%d 的绝对值是 %lf\n", b, fabs(b));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
1234 的绝对值是 1234.000000
-344 的绝对值是 344.000000
```

C 库函数 - floor()

描述

C 库函数 **double floor(double x)** 返回小于或等于 **x** 的最大的整数值。

声明

下面是 floor() 函数的声明。

```
double floor(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回不大于 x 的最大整数值。

实例

下面的实例演示了 floor() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   float val1, val2, val3, val4;

   val1 = 1.6;
   val2 = 1.2;
   val3 = 2.8;
   val4 = 2.3;

   printf("Value1 = %.1lf\n", floor(val1));
   printf("Value2 = %.1lf\n", floor(val2));
   printf("Value3 = %.1lf\n", floor(val3));
   printf("Value4 = %.1lf\n", floor(val4));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Value1 = 1.0
Value2 = 1.0
Value3 = 2.0
Value4 = 2.0
```

C 库函数 - fmod()

描述

C 库函数 **double fmod(double x, double y)** 返回 **x** 除以 **y** 的余数。

声明

下面是 fmod() 函数的声明。

```
double fmod(double x, double y)
```

参数

- **x** -- 代表分子的浮点值。
- **y** -- 代表分母的浮点值。

返回值

该函数返回 x/y 的余数。

实例

下面的实例演示了 fmod() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   float a, b;
   int c;
   a = 9.2;
   b = 3.7;
   c = 2;
   printf("%f / %d 的余数是 %lf\n", a, c, fmod(a,c));
   printf("%f / %f 的余数是 %lf\n", a, b, fmod(a,b));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
9.200000 / 2 的余数是 1.200000
9.200000 / 3.700000 的余数是 1.800000
```

C 标准库 - <setjmp.h>

简介

setjmp.h 头文件定义了宏 **setjmp()**、函数 **longjmp()** 和变量类型 **jmp_buf**，该变量类型会绕过正常的函数调用和返回规则。

库变量

下面列出了头文件 setjmp.h 中定义的变量：

变量	描述
jmp_buf	这是一个用于存储宏 **setjmp()** 和函数 **longjmp()** 相关信息的数组类型。

库宏

下面是这个库中定义的唯一的一个宏：

宏	描述
int setjmp(jmp_buf environment)	这个宏把当前环境保存在变量 **environment** 中，以便函数 **longjmp()** 后续使用。如果这个宏直接从宏调用中返回，则它会返回零，但是如果它从 **longjmp()** 函数调用中返回，则它会返回一个非零值。

库函数

下面是头文件 setjmp.h 中定义的唯一的一个函数：

函数	描述
void longjmp(jmp_buf environment, int value)	该函数恢复最近一次调用 **setjmp()** 宏时保存的环境，**jmp_buf** 参数的设置是由之前调用 setjmp() 生成的。

C 库宏 - setjmp()

描述

C 库宏 **int setjmp(jmp_buf environment)** 把当前环境保存在变量 environment 中，以便函数 **longjmp()** 后续使用。如果这个宏直接从宏调用中返回，则它会返回零，但是如果它从 **longjmp()** 函数调用中返回，则它会返回一个传给 longjmp 作为第二个参数的非零值。

声明

下面是 setjmp() 宏的声明。

```
int setjmp(jmp_buf environment)
```

参数

- **environment** -- 这是一个类型为 jmp_buf 的用于存储环境信息的对象。

返回值

这个宏可能不只返回一次。第一次，在直接调用它时，它总是返回零。当调用 longjmp 时带有设置的环境信息，这个宏会再次返回，此时它返回的值会传给 longjmp 作为第二个参数。

实例

下面的实例演示了 setjmp() 宏的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

int main()
{
   int val;
   jmp_buf env_buffer;

   /* 保存 longjmp 的调用环境 */
   val = setjmp( env_buffer );
   if( val != 0 )
   {
      printf("从 longjmp() 返回值 = %s\n", val);
      exit(0);
   }
   printf("跳转函数调用\n");
   jmpfunction( env_buffer );

   return(0);
}

void jmpfunction(jmp_buf env_buf)
{
   longjmp(env_buf, "w3cschool.cc");
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
跳转函数调用
从 longjmp() 返回值 = w3cschool.cc
```

C 库函数 - longjmp()

描述

C 库函数 **void longjmp(jmp_buf environment, int value)** 恢复最近一次调用 **setjmp()** 宏时保存的环境，**jmp_buf** 参数的设置是由之前调用 setjmp() 生成的。

声明

下面是 longjmp() 函数的声明。

```
void longjmp(jmp_buf environment, int value)
```

参数

- **environment** -- 这是一个类型为 **jmp_buf** 的对象，包含了调用 setjmp 时存储的环境信息。
- **value** -- 这是 **setjmp** 表达式要判断的值。

返回值

该函数不返回任何值。

实例

下面的实例演示了 longjmp() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

int main()
{
   int val;
   jmp_buf env_buffer;

   /* 保存 longjmp 的调用环境 */
   val = setjmp( env_buffer );
   if( val != 0 )
   {
      printf("从 longjmp() 返回值 = %s\n", val);
      exit(0);
   }
   printf("跳转函数调用\n");
   jmpfunction( env_buffer );

   return(0);
}

void jmpfunction(jmp_buf env_buf)
{
   longjmp(env_buf, "w3cschool.cc");
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
跳转函数调用
从 longjmp() 返回值 = w3cschool.cc
```

C 标准库 - <signal.h>

简介

signal.h 头文件定义了一个变量类型 **sig_atomic_t**、两个函数调用和一些宏来处理程序执行期间报告的不同信号。

库变量

下面是头文件 signal.h 中定义的变量类型：

变量	描述
sig_atomic_t	这是 **int** 类型，在信号处理程序中作为变量使用。它是一个对象的整数类型，该对象可以作为一个原子实体访问，即使存在异步信号时，该对象可以作为一个原子实体访问。

库宏

下面是头文件 signal.h 中定义的宏，这些宏将在下列两个函数中使用。**SIG_** 宏与 signal 函数一起使用来定义信号的功能。

宏	描述
SIG_DFL	默认的信号处理程序。
SIG_ERR	表示一个信号错误。
SIG_IGN	忽视信号。

SIG 宏用于表示以下各种条件的信号号码：

宏	描述
SIGABRT	程序异常终止。
SIGFPE	算术运算出错，如除数为 0 或溢出。
SIGILL	非法函数映象，如非法指令。
SIGINT	中断信号，如 ctrl-C。
SIGSEGV	非法访问存储器，如访问不存在的内存单元。
SIGTERM	发送给本程序的终止请求信号。

库函数

下面是头文件 signal.h 中定义的函数：

函数	描述
void (*signal(int sig, void (*func)(int)))(int)	这个函数设置一个函数来处理信号，即信号处理程序。
int raise(int sig)	这个函数会促使生成信号 **sig**。sig 参数与 SIG 宏兼容。

C 库函数 - signal()

描述

C 库函数 **void (*signal(int sig, void (*func)(int)))(int)** 设置一个函数来处理信号，即带有 **sig** 参数的信号处理程序。

声明

下面是 signal() 函数的声明。

```
void (*signal(int sig, void (*func)(int)))(int)
```

参数

- **sig** -- 在信号处理程序中作为变量使用的信号码。下面是一些重要的标准信号常量：

宏	信号
SIGABRT	(Signal Abort) 程序异常终止。
SIGFPE	(Signal Floating-Point Exception) 算术运算出错，如除数为 0 或溢出（不一定是浮点运算）。
SIGILL	(Signal Illegal Instruction) 非法函数映象，如非法指令，通常是由于代码中的某个变体或者尝试执行数据导致的。
SIGINT	(Signal Interrupt) 中断信号，如 ctrl-C，通常由用户生成。
SIGSEGV	(Signal Segmentation Violation) 非法访问存储器，如访问不存在的内存单元。
SIGTERM	(Signal Terminate) 发送给本程序的终止请求信号。

- **func** -- 一个指向函数的指针。它可以是一个由程序定义的函数，也可以是下面预定义函数之一：

SIG_DFL	默认的信号处理程序。
SIG_IGN	忽视信号。

返回值

该函数返回信号处理程序之前的值，当发生错误时返回 SIG_ERR。

实例

下面的实例演示了 signal() 函数的用法。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void sighandler(int);

int main()
{
   signal(SIGINT, sighandler);

   while(1)
   {
      printf("开始休眠一秒钟...\n");
      sleep(1);
   }

   return(0);
}

void sighandler(int signum)
{
   printf("捕获信号 %d，跳出...\n", signum);
   exit(1);
}
```

让我们编译并运行上面的程序，这将产生以下结果，且程序会进入无限循环，需使用 CTRL + C 键跳出程序。

```
开始休眠一秒钟...
开始休眠一秒钟...
开始休眠一秒钟...
开始休眠一秒钟...
开始休眠一秒钟...
捕获信号 2，跳出...
```

C 库函数 - raise()

描述

C 库函数 **int raise(int sig)** 会促使生成信号 **sig**。**sig** 参数与 SIG 宏兼容。

声明

下面是 raise() 函数的声明。

```
int raise(int sig)
```

参数

- **sig** -- 要发送的信号码。下面是一些重要的标准信号常量：

宏	信号
SIGABRT	(Signal Abort) 程序异常终止。
SIGFPE	(Signal Floating-Point Exception) 算术运算出错，如除数为 0 或溢出（不一定是浮点运算）。
SIGILL	(Signal Illegal Instruction) 非法函数映象，如非法指令，通常是由于代码中的某个变体或者尝试执行数据导致的。
SIGINT	(Signal Interrupt) 中断信号，如 ctrl-C，通常由用户生成。
SIGSEGV	(Signal Segmentation Violation) 非法访问存储器，如访问不存在的内存单元。
SIGTERM	(Signal Terminate) 发送给本程序的终止请求信号。

返回值

如果成功该函数返回零，否则返回非零。

实例

下面的实例演示了 raise() 函数的用法。

```
#include <signal.h>
#include <stdio.h>

void signal_catchfunc(int);

int main()
{
    int ret;

   ret = signal(SIGINT, signal_catchfunc);

   if( ret == SIG_ERR)
   {
      printf("错误：不能设置信号处理程序。\n");
      exit(0);
   }
   printf("开始生成一个信号\n");
   ret = raise(SIGINT);
   if( ret !=0 )
   {
      printf("错误：不能生成 SIGINT 信号。\n");
      exit(0);
   }

   printf("退出...\n");
   return(0);
}

void signal_catchfunc(int signal)
{
   printf("!! 信号捕获 !!\n");
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
开始生成一个信号
!! 信号捕获 !!
退出...
```

C 标准库 - <stdarg.h>

简介

stdarg.h 头文件定义了一个变量类型 **va_list** 和三个宏，这三个宏可用于在参数个数未知（即参数个数可变）时获取函数中的参数。

可变参数的函数通在参数列表的末尾是使用省略号(,...)定义的。

库变量

下面是头文件 stdarg.h 中定义的变量类型：

变量	描述
va_list	这是一个适用于 **va_start()、va_arg()** 和 **va_end()** 这三个宏存储信息的类型。

库宏

下面是头文件 stdarg.h 中定义的宏：

宏	描述
void va_start(va_list ap, last_arg)	这个宏初始化 **ap** 变量，它与 **va_arg** 和 **va_end** 宏是一起使用的。**last_arg** 是最后一个传递给函数的已知的固定参数，即省略号之前的参数。
type va_arg(va_list ap, type)	这个宏检索函数参数列表中类型为 **type** 的下一个参数。
void va_end(va_list ap)	这个宏允许使用了 **va_start** 宏的带有可变参数的函数返回。如果在从函数返回之前没有调用 **va_end**，则结果为未定义。

C 库宏 - va_start()

描述

C 库宏 **void va_start(va_list ap, last_arg)** 初始化 **ap** 变量，它与 **va_arg** 和 **va_end** 宏是一起使用的。**last_arg** 是最后一个传递给函数的已知的固定参数，即省略号之前的参数。

这个宏必须在使用 **va_arg** 和 **va_end** 之前被调用。

声明

下面是 va_start() 宏的声明。

```
void va_start(va_list ap, last_arg);
```

参数

- **ap** -- 这是一个 **va_list** 类型的对象，它用来存储通过 **va_arg** 获取额外参数时所必需的信息。
- **last_arg** -- 最后一个传递给函数的已知的固定参数。

返回值

NA

实例

下面的实例演示了 va_start() 宏的用法。

```
#include<stdarg.h>
#include<stdio.h>

int sum(int, ...);

int main(void)
{
   printf("10、20 和 30 的和 = %d\n",  sum(3, 10, 20, 30) );
   printf("4、20、25 和 30 的和 = %d\n",  sum(4, 4, 20, 25, 30) );

   return 0;
}

int sum(int num_args, ...)
{
   int val = 0;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val += va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
10、20 和 30 的和 = 60
4、20、25 和 30 的和 = 79
```

C 库宏 - va_arg()

描述

C 库宏 **type va_arg(va_list ap, type)** 检索函数参数列表中类型为 **type** 的下一个参数。它无法判断检索到的参数是否是传给函数的最后一个参数。

声明

下面是 va_arg() 宏的声明。

```
type va_arg(va_list ap, type)
```

参数

- **ap** -- 这是一个 **va_list** 类型的对象，存储了有关额外参数和检索状态的信息。该对象应在第一次调用 va_arg 之前通过调用 va_start 进行初始化。
- **type** -- 这是一个类型名称。该类型名称是作为扩展自该宏的表达式的类型来使用的。

返回值

该宏返回下一个额外的参数，是一个类型为 type 的表达式。

实例

下面的实例演示了 va_arg() 宏的用法。

```
#include <stdarg.h>
#include <stdio.h>

int sum(int, ...);

int main()
{
   printf("15 和 56 的和 = %d\n",  sum(2, 15, 56) );
   return 0;
}

int sum(int num_args, ...)
{
   int val = 0;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val += va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
15 和 56 的和 = 71
```

C 库宏 - va_end()

描述

C 库宏 **void va_end(va_list ap)** 允许使用了 **va_start** 宏的带有可变参数的函数返回。如果在从函数返回之前没有调用 **va_end**，则结果为未定义。

声明

下面是 va_end() 宏的声明。

```
void va_end(va_list ap)
```

参数

- **ap** -- 这是之前由同一函数中的 va_start 初始化的 va_list 对象。

返回值

该宏不返回任何值。

实例

下面的实例演示了 va_end() 宏的用法。

```
#include <stdarg.h>
#include <stdio.h>

int mul(int, ...);

int main()
{
   printf("15 * 12 = %d\n",  mul(2, 15, 12) );

   return 0;
}

int mul(int num_args, ...)
{
   int val = 1;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val *= va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
15 * 12 =  180
```

C 标准库 - <stddef.h>

简介

stddef .h 头文件定义了各种变量类型和宏。这些定义中的大部分也出现在其它头文件中。

库变量

下面是头文件 stddef.h 中定义的变量类型：

变量	描述
ptrdiff_t	这是有符号整数类型，它是两个指针相减的结果。
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。
wchar_t	这是一个宽字符常量大小的整数类型。

库宏

下面是头文件 stddef.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
offsetof(type, member-designator)	这会生成一个类型为 size*t* 的整型常量，它是一个结构成员相对于结构开头的字节偏移量。成员是由 _*member-designator* 给定的，结构的名称是在 *type* 中给定的。

C 库宏 - NULL

描述

C 库宏 **NULL** 是一个空指针常量的值。它可以被定义为 **((void*)0), 0** 或 **0L**，这取决于编译器供应商。

声明

下面是取决于编译器的 NULL 宏的声明。

```
#define NULL ((char *)0)

或

#define NULL 0L

或

#define NULL 0
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 NULL 宏的用法。

```
#include <stddef.h>
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "r");
   if( fp != NULL )
   {
      printf("成功打开文件 file.txt\n");
      fclose(fp);
   }

   fp = fopen("nofile.txt", "r");
   if( fp == NULL )
   {
      printf("不能打开文件 nofile.txt\n");
   }

   return(0);
}
```

假设文件 **file.txt** 已存在，但是 **nofile.txt** 不存在。让我们编译并运行上面的程序，这将产生以下结果：

```
成功打开文件 file.txt
不能打开文件 nofile.txt
```

C 库宏 - offsetof()

描述

C 库宏 **offsetof(type, member-designator)** 会生成一个类型为 **size_t** 的整型常量，它是一个结构成员相对于结构开头的字节偏移量。成员是由 member-designator 给定的，结构的名称是在 type 中给定的。

声明

下面是 offsetof() 宏的声明。

```
offsetof(type, member-designator)
```

参数

- **type** -- 这是一个 class 类型，其中，member-designator 是一个有效的成员指示器。
- **member-designator** -- 这是一个 class 类型的成员指示器。

返回值

该宏返回类型为 **size_t** 的值，表示 type 中成员的偏移量。

实例

下面的实例演示了 offsetof() 宏的用法。

```
#include <stddef.h>
#include <stdio.h>

struct address {
   char name[50];
   char street[50];
   int phone;
};

int main()
{
   printf("address 结构中的 name 偏移 = %d 字节。\n",
   offsetof(struct address, name));

   printf("address 结构中的 street 偏移 = %d 字节。\n",
   offsetof(struct address, street));

   printf("address 结构中的 phone 偏移 = %d 字节。\n",
   offsetof(struct address, phone));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
address 结构中的 name 偏移 = 0 字节。
address 结构中的 street 偏移 = 50 字节。
address 结构中的 phone 偏移 = 100 字节。
```

C 标准库 - <stdio.h>

简介

stdio .h 头文件定义了三个变量类型、一些宏和各种函数来执行输入和输出。

库变量

下面是头文件 stdio.h 中定义的变量类型：

变量	描述
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。
FILE	这是一个适合存储文件流信息的对象类型。
fpos_t	这是一个适合存储文件中任何位置的对象类型。

库宏

下面是头文件 stdio.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
_IOFBF、_IOLBF 和 _IONBF	这些宏扩展了带有特定值的整型常量表达式，并适用于 **setvbuf** 函数的第三个参数。
BUFSIZ	这个宏是一个整数，该整数代表了 **setbuf** 函数使用的缓冲区大小。
EOFM	这个宏是一个表示已经到达文件结束的负整数。
FOPEN_MAX	这个宏是一个整数，该整数代表了系统可以同时打开的文件数量。
FILENAME_MAX	这个宏是一个整数，该整数代表了字符数组可以存储的文件名的最大长度。如果实现没有任何限制，则该值应为推荐的最大值。
L_tmpnam	这个宏是一个整数，该整数代表了字符数组可以存储的由 tmpnam 函数创建的临时文件名的最大长度。
SEEK_CUR、SEEK_END 和 SEEK_SET	这些宏是在These macros are used in the **fseek** 函数中使用，用于在一个文件中定位不同的位置。
TMP_MAX	这个宏是 tmpnam 函数可生成的独特文件名的最大数量。
stderr、stdin 和 stdout	这些宏是指向 FILE 类型的指针，分别对应于标准错误、标准输入和标准输出流。

库函数

下面是头文件 stdio.h 中定义的函数：

> 为了更好地理解函数，请按照下面的序列学习这些函数，因为第一个函数中创建的文件会在后续的函数中使用到。

函数	描述
int fclose(FILE *stream)	关闭流 stream。刷新所有的缓冲区。
void clearerr(FILE *stream)	清除给定流 stream 的文件结束和错误标识符。
int feof(FILE *stream)	测试给定流 stream 的文件结束标识符。
int ferror(FILE *stream)	测试给定流 stream 的错误标识符。
int fflush(FILE *stream)	刷新流 stream 的输出缓冲区。
int fgetpos(FILE *stream, fpos_t *pos)	获取流 stream 的当前文件位置，并把它写入到 pos。
FILE *fopen(const char	

*filename, const char *mode)	使用给定的模式 mode 打开 filename 所指向的文件。
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)	从给定流 stream 读取数据到 ptr 所指向的数组中。
FILE *freopen(const char *filename, const char *mode, FILE *stream)	把一个新的文件名 filename 与给定的打开的流 stream 关联，同时关闭流中的旧文件。
int fseek(FILE *stream, long int offset, int whence)	设置流 stream 的文件位置为给定的偏移 offset，参数 *offset* 意味着从给定的 *whence* 位置查找的字节数。
int fsetpos(FILE *stream, const fpos_t *pos)	设置给定流 stream 的文件位置为给定的位置。参数 *pos* 是由函数 fgetpos 给定的位置。
long int ftell(FILE *stream)	返回给定流 stream 的当前文件位置。
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)	把 ptr 所指向的数组中的数据写入到给定流 stream 中。
int remove(const char *filename)	删除给定的文件名 filename，以便它不再被访问。
int rename(const char *old_filename, const char *new_filename)	把 old_filename 所指向的文件名改为 new_filename。
void rewind(FILE *stream)	设置文件位置为给定流 stream 的文件的开头。
void setbuf(FILE *stream, char *buffer)	定义流 stream 应如何缓冲。
int setvbuf(FILE *stream, char *buffer, int mode, size_t size)	另一个定义流 stream 应如何缓冲的函数。
FILE *tmpfile(void)	以二进制更新模式(wb+)创建临时文件。
char *tmpnam(char *str)	生成并返回一个有效的临时文件名，该文件名之前是不存在的。
int fprintf(FILE *stream, const char *format, ...)	发送格式化输出到流 stream 中。
int printf(const char *format, ...)	发送格式化输出到标准输出 stdout。
int sprintf(char *str, const char *format, ...)	发送格式化输出到字符串。

int vfprintf(FILE *stream, const char *format, va_list arg)	使用参数列表发送格式化输出到流 stream 中。
int vprintf(const char *format, va_list arg)	使用参数列表发送格式化输出到标准输出 stdout。
int vsprintf(char *str, const char *format, va_list arg)	使用参数列表发送格式化输出到字符串。
int fscanf(FILE *stream, const char *format, ...)	从流 stream 读取格式化输入。
int scanf(const char *format, ...)	从标准输入 stdin 读取格式化输入。
int sscanf(const char *str, const char *format, ...)	从字符串读取格式化输入。
int fgetc(FILE *stream)	从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。
char *fgets(char *str, int n, FILE *stream)	从指定的流 stream 读取一行，并把它存储在 str 所指向的字符串内。当读取 **(n-1)** 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。
int fputc(int char, FILE *stream)	把参数 char 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。
int fputs(const char *str, FILE *stream)	把字符串写入到指定的流 stream 中，但不包括空字符。
int getc(FILE *stream)	从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。
int getchar(void)	从标准输入 stdin 获取一个字符（一个无符号字符）。
char *gets(char *str)	从标准输入 stdin 读取一行，并把它存储在 str 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。
int putc(int char, FILE *stream)	把参数 char 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。
int putchar(int char)	把参数 char 指定的字符（一个无符号字符）写入到标准输出 stdout 中。
int puts(const char *str)	把一个字符串写入到标准输出 stdout，直到空字符，但不包括空字符。换行符会被追加到输出中。
int ungetc(int char, FILE *stream)	把字符 char（一个无符号字符）推入到指定的流 stream 中，以便它是下一个被读取到的字符。
void perror(const char *str)	把一个描述性错误消息输出到标准错误 stderr。首先输出字符串 str，后跟一个冒号，然后是一个空格。

C 库函数 - fclose()

描述

C 库函数 **int fclose(FILE *stream)** 关闭流 stream。刷新所有的缓冲区。

声明

下面是 fclose() 函数的声明。

```
int fclose(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了要被关闭的流。

返回值

如果流成功关闭，则该方法返回零。如果失败，则返回 EOF。

实例

下面的实例演示了 fclose() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;

   fp = fopen("file.txt", "w");

   fprintf(fp, "%s", "这里是 w3cschool.cc");
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，然后写入下面的文本行，最后使用 **fclose()** 函数关闭文件。

```
这里是 w3cschool.cc
```

C 库函数 - clearerr()

描述

C 库函数 **void clearerr(FILE *stream)** 清除给定流 stream 的文件结束和错误标识符。

声明

下面是 clearerr() 函数的声明。

```
void clearerr(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

这不会失败，且不会设置外部变量 errno，但是如果它检测到它的参数不是一个有效的流，则返回 -1，并设置 errno 为 EBADF。

实例

下面的实例演示了 clearerr() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char c;

   fp = fopen("file.txt", "w");

   c = fgetc(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   clearerr(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它是一个空文件。让我们编译并运行上面的程序，因为我们试图读取一个以只写模式打开的文件，这将产生以下结果。

```
读取文件：file.txt 时发生错误
```

C 库函数 - feof()

描述

C 库函数 **int feof(FILE *stream)** 测试给定流 stream 的文件结束标识符。

声明

下面是 feof() 函数的声明。

```
int feof(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

当设置了与流关联的文件结束标识符时，该函数返回一个非零值，否则返回零。

实例

下面的实例演示了 feof() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   if(fp == NULL)
   {
      perror("打开文件时发生错误");
      return(-1);
   }
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下所示。该文件将作为我们实例程序中的一个输入使用：

```
这里是 w3cschool.cc
```

让我们编译并运行上面的程序，这将产生以下结果：

```
这里是 w3cschool.cc
```

C 库函数 - ferror()

描述

C 库函数 **int ferror(FILE *stream)** 测试给定流 stream 的错误标识符。

声明

下面是 ferror() 函数的声明。

```
int ferror(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

如果设置了与流关联的错误标识符，该函数返回一个非零值，否则返回一个零值。

实例

下面的实例演示了 ferror() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char c;

   fp = fopen("file.txt", "w");

   c = fgetc(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   clearerr(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它是一个空文件。让我们编译并运行上面的程序，因为我们试图读取一个以只写模式打开的文件，这将产生以下结果。

```
读取文件：file.txt 时发生错误
```

C 库函数 - fflush()

描述

C 库函数 **int fflush(FILE *stream)** 刷新流 stream 的输出缓冲区。

声明

下面是 fflush() 函数的声明。

```
int fflush(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个缓冲流。

返回值

如果成功，该函数返回零值。如果发生错误，则返回 EOF，且设置错误标识符（即 feof）。

实例

下面的实例演示了 fflush() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{

   char buff[1024];

   memset( buff, '\0', sizeof( buff ));

   fprintf(stdout, "启用全缓冲\n");
   setvbuf(stdout, buff, _IOFBF, 1024);

   fprintf(stdout, "这里是 w3cschool.cc\n");
   fprintf(stdout, "该输出将保存到 buff\n");
   fflush( stdout );

   fprintf(stdout, "这将在编程时出现\n");
   fprintf(stdout, "最后休眠五秒钟\n");

   sleep(5);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果。在这里，程序把缓冲输出保存到 **buff**，直到首次调用 **fflush()** 为止，然后开始缓冲输出，最后休眠 5 秒钟。它会在程序结束之前，发送剩余的输出到 STDOUT。

```
启用全缓冲
这里是 w3cschool.cc
该输出将保存到 buff
这将在编程时出现
最后休眠五秒钟
```

C 库函数 - fgetpos()

描述

C 库函数 **int fgetpos(FILE *stream, fpos_t *pos)** 获取流 **stream** 的当前文件位置，并把它写入到 **pos**。

声明

下面是 fgetpos() 函数的声明。

```
int fgetpos(FILE *stream, fpos_t *pos)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **pos** -- 这是指向 fpos_t 对象的指针。

返回值

如果成功，该函数返回零。如果发生错误，则返回非零值。

实例

下面的实例演示了 fgetpos() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("这将覆盖之前的内容", fp);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，它的内容如下。首先我们使用 **fgetpos()** 函数获取文件的初始位置，接着我们向文件写入 *Hello, World!*，然后我们使用 **fsetpos()** 函数来重置写指针到文件的开头，重写文件为下列内容：

```
这将覆盖之前的内容
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   int n = 0;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }

   fclose(fp);

   return(0);
}
```

C 库函数 - fopen()

描述

C 库函数 **FILE *fopen(const char* filename, const char *mode)** 使用给定的模式 **mode** 打开 **filename** 所指向的文件。

声明

下面是 fopen() 函数的声明。

```
FILE *fopen(const char *filename, const char *mode)
```

参数

- **filename** -- 这是 C 字符串，包含了要打开的文件名称。
- **mode** -- 这是 C 字符串，包含了文件访问模式，模式如下：

模式	描述
"r"	打开一个用于读取的文件。该文件必须存在。
"w"	创建一个用于写入的空文件。如果文件名称与已存在的文件相同，则会删除已有文件的内容，文件被视为一个新的空文件。
"a"	追加到一个文件。写操作向文件末尾追加数据。如果文件不存在，则创建文件。
"r+"	打开一个用于更新的文件，可读取也可写入。该文件必须存在。
"w+"	创建一个用于读写的空文件。
"a+"	打开一个用于读取和追加的文件。

返回值

该函数返回一个 FILE 指针。否则返回 NULL，且设置全局变量 errno 来标识错误。

实例

下面的实例演示了 fopen() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2014);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个带有一下内容的文件 **file.txt**：

```
We are in 2014
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fread()

描述

C 库函数 **size_t fread(void *ptr, size_t size, size_t nmemb, FILE* **stream)** 从给定流 **stream** 读取数据到 **ptr** 所指向的数组中。

声明

下面是 fread() 函数的声明。

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

参数

- **ptr** -- 这是指向带有最小尺寸 *size*nmemb* 字节的内存块的指针。
- **size** -- 这是要读取的每个元素的大小，以字节为单位。
- **nmemb** -- 这是元素的个数，每个元素的大小为 size 字节。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个输入流。

返回值

成功读取的元素总数会以 size_t 对象返回，size_t 对象是一个整型数据类型。如果总数与 nmemb 参数不同，则可能发生了一个错误或者到达了文件末尾。

实例

下面的实例演示了 fread() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   FILE *fp;
   char c[] = "This is w3cschool";
   char buffer[20];

   /* 打开文件用于读写 */
   fp = fopen("file.txt", "w+");

   /* 写入数据到文件 */
   fwrite(c, strlen(c) + 1, 1, fp);

   /* 查找文件的开头 */
   fseek(fp, SEEK_SET, 0);

   /* 读取并显示数据 */
   fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，然后写入内容 *this is w3cschool*。接下来我们使用 **fseek()** 函数来重置写指针到文件的开头，文件内容如下所示：

```
This is w3cschool
```

C 库函数 - freopen()

描述

C 库函数 **FILE *freopen(const char* filename, const char *mode, FILE* stream)** 把一个新的文件名 **filename** 与给定的打开的流 **stream** 关联，同时关闭流中的旧文件。

声明

下面是 freopen() 函数的声明。

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

参数

- **filename** -- 这是 C 字符串，包含了要打开的文件名称。
- **mode** -- 这是 C 字符串，包含了文件访问模式，模式如下：

模式	描述
"r"	打开一个用于读取的文件。该文件必须存在。
"w"	创建一个用于写入的空文件。如果文件名称与已存在的文件相同，则会删除已有文件的内容，文件被视为一个新的空文件。
"a"	追加到一个文件。写操作向文件末尾追加数据。如果文件不存在，则创建文件。
"r+"	打开一个用于更新的文件，可读取也可写入。该文件必须存在。
"w+"	创建一个用于读写的空文件。
"a+"	打开一个用于读取和追加的文件。

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被重新打开的流。

返回值

如果文件成功打开，则函数返回一个指针，指向用于标识流的对象。否则，返回空指针。

实例

下面的实例演示了 freopen() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   printf("该文本重定向到 stdout\n");

   fp = freopen("file.txt", "w+", stdout);

   printf("该文本重定向到 file.txt\n");

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将发送下列行到标准输出 STDOUT，因为起初我们并没有打开标准输出：

```
该文本重定向到 stdout
```

在调用 **freopen()** 之后，它会关联标准输出 STDOUT 到文件 **file.txt**，无论我们在标准输出 STDOUT 中写了什么都会被写入 file.txt，所以文件 file.txt 将有以下内容。

```
该文本重定向到 file.txt
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fseek()

描述

C 库函数 **int fseek(FILE *stream, long int offset, int whence)** 设置流 **stream** 的文件位置为给定的偏移 **offset**，参数 offset 意味着从给定的 **whence** 位置查找的字节数。

声明

下面是 fseek() 函数的声明。

```
int fseek(FILE *stream, long int offset, int whence)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **offset** -- 这是相对 whence 的偏移量，以字节为单位。
- **whence** -- 这是表示开始添加偏移 offset 的位置。它一般指定为下列常量之一：

常量	描述
SEEK_SET	文件的开头
SEEK_CUR	文件指针的当前位置
SEEK_END	文件的末尾

返回值

如果成功，则该函数返回零，否则返回非零值。

实例

下面的实例演示了 fseek() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","w+");
   fputs("This is w3cschool.cc", fp);

   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Langauge", fp);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建文件 **file.txt**，它的内容如下。最初程序创建文件和写入 *This is w3cschool.cc*，但是之后我们在第七个位置重置了写指针，并使用 puts() 语句来重写文件，内容如下：

```
This is C Programming Langauge
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fsetpos()

描述

C 库函数 **int fsetpos(FILE *stream, const fpos_t *pos)** 设置给定流 **stream** 的文件位置为给定的位置。参数 **pos** 是由函数 fgetpos 给定的位置。

声明

下面是 fsetpos() 函数的声明。

```
int fsetpos(FILE *stream, const fpos_t *pos)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **pos** -- 这是指向 fpos_t 对象的指针，该对象包含了之前通过 fgetpos 获得的位置。

返回值

如果成功，该函数返回零值，否则返回非零值，并设置全局变量 **errno** 为一个正值，该值可通过 perror 来解释。

实例

下面的实例演示了 fsetpos() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("这将覆盖之前的内容", fp);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，它的内容如下。首先我们使用 **fgetpos()** 函数获取文件的初始位置，接着我们向文件写入 *Hello, World!*，然后我们使用 **fsetpos()** 函数来重置写指针到文件的开头，重写文件为下列内容：

```
这将覆盖之前的内容
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - ftell()

描述

C 库函数 **long int ftell(FILE *stream)** 返回给定流 stream 的当前文件位置。

声明

下面是 ftell() 函数的声明。

```
long int ftell(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

该函数返回位置标识符的当前值。如果发生错误，则返回 -1L，全局变量 errno 被设置为一个正值。

实例

下面的实例演示了 ftell() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int len;

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror ("打开文件错误");
      return(-1);
   }
   fseek(fp, 0, SEEK_END);

   len = ftell(fp);
   fclose(fp);

   printf("file.txt 的总大小 = %d 字节\n", len);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下：

```
This is w3cschool.cc
```

让我们编译并运行上面的程序，如果文件内容如上所示，这将产生以下结果，否则会根据文件内容给出不同的结果：

```
file.txt 的总大小 = 21 字节
```

C 库函数 - fwrite()

描述

C 库函数 **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE* stream)** 把 **ptr** 所指向的数组中的数据写入到给定流 **stream** 中。

声明

下面是 fwrite() 函数的声明。

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

参数

- **ptr** -- 这是指向要被写入的元素数组的指针。
- **size** -- 这是要被写入的每个元素的大小，以字节为单位。
- **nmemb** -- 这是元素的个数，每个元素的大小为 size 字节。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个输出流。

返回值

如果成功，该函数返回一个 size_t 对象，表示元素的总数，该对象时一个整型数据类型。如果该数字与 nmemb 参数不同，则会显示一个错误。

实例

下面的实例演示了 fwrite() 函数的用法。

```
#include<stdio.h>

int main ()
{
   FILE *fp;
   char str[] = "This is w3cschool.cc";

   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，它的内容如下：

```
This is w3cschool.cc
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - remove()

描述

C 库函数 **int remove(const char *filename)** 删除给定的文件名 **filename**，以便它不再被访问。

声明

下面是 remove() 函数的声明。

```
int remove(const char *filename)
```

参数

- **filename** -- 这是 C 字符串，包含了要被删除的文件名称。

返回值

如果成功，则返回零。如果错误，则返回 -1，并设置 errno。

实例

下面的实例演示了 remove() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int ret;
   FILE *fp;
   char filename[] = "file.txt";

   fp = fopen(filename, "w");

   fprintf(fp, "%s", "这里是 w3cschool.cc");
   fclose(fp);

   ret = remove(filename);

   if(ret == 0)
   {
      printf("文件删除成功");
   }
   else
   {
      printf("错误：不能删除该文件");
   }

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。我们将使用上面的程序来删除该文件。让我们编译并运行上面的程序，这将生成下面的消息，且文件被永久删除。

```
文件删除成功
```

C 库函数 - rename()

描述

C 库函数 **int rename(const char *old_filename, const char *new_filename)** 把 **old_filename** 所指向的文件名改为 **new_filename**。

声明

下面是 rename() 函数的声明。

```
int rename(const char *old_filename, const char *new_filename)
```

参数

- **old_filename** -- 这是 C 字符串，包含了要被重命名/移动的文件名称。
- **new_filename** -- 这是 C 字符串，包含了文件的新名称。

返回值

如果成功，则返回零。如果错误，则返回 -1，并设置 errno。

实例

下面的实例演示了 rename() 函数的用法。

```
#include <stdio.h>

int main ()
{
   int ret;
   char oldname[] = "file.txt";
   char newname[] = "newfile.txt";

   ret = rename(oldname, newname);

   if(ret == 0)
   {
      printf("文件重命名成功");
   }
   else
   {
      printf("错误：不能重命名该文件");
   }

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。我们将使用上面的程序来重命名该文件。让我们编译并运行上面的程序，这将生成下面的消息，且文件被重命名为 **newfile.txt** 文件。

```
文件重命名成功
```

C 库函数 - rewind()

描述

C 库函数 **void rewind(FILE *stream)** 设置文件位置为给定流 **stream** 的文件的开头。

声明

下面是 rewind() 函数的声明。

```
void rewind(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

该函数不返回任何值。

实例

下面的实例演示了 rewind() 函数的用法。

```
#include <stdio.h>

int main()
{
   char str[] = "This is w3cschool.cc";
   FILE *fp;
   int ch;

   /* 首先让我们在文件中写入一些内容 */
   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );
   fclose(fp);

   fp = fopen( "file.txt" , "r" );
   while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);
   }
   rewind(fp);
   printf("\n");
   while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);

   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下：

```
This is w3cschool.cc
```

让我们编译并运行上面的程序，这将产生以下结果：

```
This is w3cschool.cc
This is w3cschool.cc
```

C 库函数 - setbuf()

描述

C 库函数 **void setbuf(FILE *stream, char* buffer)** 定义流 stream 应如何缓冲。该函数应在与流 stream 相关的文件被打开时，且还未发生任何输入或输出操作之前被调用一次。

声明

下面是 setbuf() 函数的声明。

```
void setbuf(FILE *stream, char *buffer)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了一个打开的流。
- **buffer** -- 这是分配给用户的缓冲，它的长度至少为 BUFSIZ 字节，BUFSIZ 是一个宏常量，表示数组的长度。

返回值

该函数不返回任何值。

实例

下面的实例演示了 setbuf() 函数的用法。

```
#include <stdio.h>

int main()
{
   char buf[BUFSIZ];

   setbuf(stdout, buf);
   puts("This is w3cschool");

   fflush(stdout);
   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果。在这里，程序在即将输出的时候，发送输出到 STDOUT，否则它将缓冲输出。您也可以使用 fflush() 函数来 to 刷新输出。

```
This is w3cschool
```

C 库函数 - tmpfile()

描述

C 库函数 **FILE *tmpfile(void)** 以二进制更新模式(wb+)创建临时文件。被创建的临时文件会在流关闭的时候或者在程序终止的时候自动删除。

声明

下面是 tmpfile() 函数的声明。

```
FILE *tmpfile(void)
```

参数

- **NA**

返回值

如果成功，该函数返回一个指向被创建的临时文件的流指针。如果文件未被创建，则返回 NULL。

实例

下面的实例演示了 tmpfile() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = tmpfile();
   printf("临时文件被创建\n");

   /* 您可以在这里使用临时文件 */

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，它将在 /tmp 文件夹中创建一个临时文件，但是一旦程序退出，临时文件会被自动删除，且程序会产生以下结果：

```
临时文件被创建
```

C 库函数 - tmpnam()

描述

C 库函数 **char *tmpnam(char *str)** 生成并返回一个有效的临时文件名，该文件名之前是不存在的。如果 **str** 为空，则只会返回临时文件名。

声明

下面是 tmpnam() 函数的声明。

```
char *tmpnam(char *str)
```

参数

- **str** -- 这是一个指向字符数组的指针，其中，临时文件名将被存储为 C 字符串。

返回值

- 一个指向 C 字符串的指针，该字符串存储了临时文件名。如果 str 是一个空指针，则该指针指向一个内部缓冲区，缓冲区在下一次调用函数时被覆盖。
- 如果 str 不是一个空指针，则返回 str。如果函数未能成功创建可用的文件名，则返回一个空指针。

实例

下面的实例演示了 tmpnam() 函数的用法。

```
#include <stdio.h>

int main()
{
   char buffer[L_tmpnam];
   char *ptr;

   tmpnam(buffer);
   printf("临时名称 1: %s\n", buffer);

   ptr = tmpnam(NULL);
   printf("临时名称 2: %s\n", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
临时名称 1: /tmp/filebaalTb
临时名称 2: /tmp/filedCIbb0
```

C 库函数 - fprintf()

描述

C 库函数 **int fprintf(FILE *stream, const char *format, ...)** 发送格式化输出到流 stream 中。

声明

下面是 fprintf() 函数的声明。

```
int fprintf(FILE *stream, const char *format, ...)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **format** -- 这是 C 字符串，包含了要被写入到流 stream 中的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 fprintf() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2014);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建文件 **file.txt**，它的内容如下：

```
We are in 2014
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - printf()

描述

C 库函数 **int printf(const char *format, ...)** 发送格式化输出到标准输出 stdout。

声明

下面是 printf() 函数的声明。

```
int printf(const char *format, ...)
```

参数

- **format** -- 这是字符串，包含了要被写入到标准输出 stdout 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 printf() 函数的用法。

```
#include <stdio.h>

int main ()
{
   int ch;

   for( ch = 75 ; ch <= 100; ch++ ) {
      printf("ASCII 值 = %d, 字符 = %c\n", ch , ch );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
ASCII 值 = 75, 字符 = K
ASCII 值 = 76, 字符 = L
ASCII 值 = 77, 字符 = M
ASCII 值 = 78, 字符 = N
ASCII 值 = 79, 字符 = O
ASCII 值 = 80, 字符 = P
ASCII 值 = 81, 字符 = Q
ASCII 值 = 82, 字符 = R
ASCII 值 = 83, 字符 = S
ASCII 值 = 84, 字符 = T
ASCII 值 = 85, 字符 = U
ASCII 值 = 86, 字符 = V
ASCII 值 = 87, 字符 = W
ASCII 值 = 88, 字符 = X
ASCII 值 = 89, 字符 = Y
ASCII 值 = 90, 字符 = Z
ASCII 值 = 91, 字符 = [
ASCII 值 = 92, 字符 = \
ASCII 值 = 93, 字符 = ]
ASCII 值 = 94, 字符 = ^
ASCII 值 = 95, 字符 = _
ASCII 值 = 96, 字符 = `
ASCII 值 = 97, 字符 = a
ASCII 值 = 98, 字符 = b
ASCII 值 = 99, 字符 = c
ASCII 值 = 100, 字符 = d
```

C 库函数 - sprintf()

描述

C 库函数 **int sprintf(char *str, const char* format, ...)** 发送格式化输出到 **str** 所指向的字符串。

声明

下面是 sprintf() 函数的声明。

```
int sprintf(char *str, const char *format, ...)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了 C 字符串。
- **format** -- 这是字符串，包含了要被写入到字符串 str 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，则返回写入的字符总数，不包括字符串追加在字符串末尾的空字符。如果失败，则返回一个负数。

实例

下面的实例演示了 sprintf() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main()
{
   char str[80];

   sprintf(str, "Pi 的值 = %f", M_PI);
   puts(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Pi 的值 = 3.141593
```

C 库函数 - vfprintf()

描述

C 库函数 **int vfprintf(FILE *stream, const char *format, va_list arg)** 使用参数列表发送格式化输出到流 stream 中。

声明

下面是 vfprintf() 函数的声明。

```
int vfprintf(FILE *stream, const char *format, va_list arg)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **format** -- 这是 C 字符串，包含了要被写入到流 stream 中的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- **arg** -- 一个表示可变参数列表的对象。这应被 <stdarg> 中定义的 va_start 宏初始化。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 vfprintf() 函数的用法。

```
#include <stdio.h>
#include <stdarg.h>

void WriteFrmtd(FILE *stream, char *format, ...)
{
   va_list args;

   va_start(args, format);
   vfprintf(stream, format, args);
   va_end(args);
}

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","w");

   WriteFrmtd(fp, "This is just one argument %d \n", 10);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将打开当前目录中的文件 **file.txt**，并写入以下内容：

```
This is just one argument 10
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - vprintf()

描述

C 库函数 **int vprintf(const char *format, va_list arg)** 使用参数列表发送格式化输出到标准输出 stdout。

声明

下面是 vprintf() 函数的声明。

```
int vprintf(const char *format, va_list arg)
```

参数

- **format** -- 这是字符串，包含了要被写入到标准输出 stdout 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- **arg** -- 一个表示可变参数列表的对象。这应被 <stdarg> 中定义的 va_start 宏初始化。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 vprintf() 函数的用法。

```
#include <stdio.h>
#include <stdarg.h>

void WriteFrmtd(char *format, ...)
{
   va_list args;

   va_start(args, format);
   vprintf(format, args);
   va_end(args);
}

int main ()
{
   WriteFrmtd("%d variable argument\n", 1);
   WriteFrmtd("%d variable %s\n", 2, "arguments");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
1 variable argument
2 variable arguments
```

C 库函数 - vsprintf()

描述

C 库函数 **int vsprintf(char *str, const char *format, va_list arg)** 使用参数列表发送格式化输出到字符串。

声明

下面是 vsprintf() 函数的声明。

```
int vsprintf(char *str, const char *format, va_list arg)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了 C 字符串。
- **format** -- 这是字符串，包含了要被写入到字符串 str 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- **arg** -- 一个表示可变参数列表的对象。这应被 <stdarg> 中定义的 va_start 宏初始化。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 vsprintf() 函数的用法。

```
#include <stdio.h>
#include <stdarg.h>

char buffer[80];
int vspfunc(char *format, ...)
{
   va_list aptr;
   int ret;

   va_start(aptr, format);
   ret = vsprintf(buffer, format, aptr);
   va_end(aptr);

   return(ret);
}

int main()
{
   int i = 5;
   float f = 27.0;
   char str[50] = "w3cschool.cc";

   vspfunc("%d %f %s", i, f, str);
   printf("%s\n", buffer);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
5 27.000000 w3cschool.cc
```

C 库函数 - fscanf()

描述

C 库函数 **int fscanf(FILE *stream, const char *format, ...)** 从流 stream 读取格式化输入。

声明

下面是 fscanf() 函数的声明。

```
int fscanf(FILE *stream, const char *format, ...)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **format** -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 *format* 说明符。
  format 说明符形式为 **[=%[*][width][modifiers]type=]**，具体讲解如下：

参数	描述
*	这是一个可选的星号，表示数据是从流 stream 中读取的，但是可以被忽视，即它不存储在对应的参数中。
width	这指定了在当前读取操作中读取的最大字符数。
modifiers	为对应的附加参数所指向的数据指定一个不同于整型（针对 d、i 和 n）、无符号整型（针对 o、u 和 x）或浮点型（针对 e、f 和 g）的大小： h ：短整型（针对 d、i 和 n），或无符号短整型（针对 o、u 和 x） l ：长整型（针对 d、i 和 n），或无符号长整型（针对 o、u 和 x），或双精度型（针对 e、f 和 g） L ：长双精度型（针对 e、f 和 g）
type	一个字符，指定了要被读取的数据类型以及数据读取方式。具体参见下一个表格。

fscanf 类型说明符：

类型	合格的输入	参数的类型
c	单个字符：读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。	char *
d	十进制整数：数字前面的 + 或 - 号是可选的。	int *
e,E,f,g,G	浮点数：包含了一个小数点、一个可选的前置符号 + 或 -、一个可选的后置字符 e 或 E，以及一个十进制数字。两个有效的实例 -732.103 和 7.12e4	float *
o	八进制整数。	int *
s	字符串。这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。	char *
u	无符号的十进制整数。	unsigned int *
x,X	十六进制整数。	int *

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 fscanf() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char str1[10], str2[10], str3[10];
   int year;
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fputs("We are in 2014", fp);

   rewind(fp);
   fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

   printf("Read String1 |%s|\n", str1 );
   printf("Read String2 |%s|\n", str2 );
   printf("Read String3 |%s|\n", str3 );
   printf("Read Integer |%d|\n", year );

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2014|
```

C 库函数 - scanf()

描述

C 库函数 **int scanf(const char *format, ...)** 从标准输入 stdin 读取格式化输入。

声明

下面是 scanf() 函数的声明。

```
int scanf(const char *format, ...)
```

参数

- **format** -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 *format* 说明符。
  format 说明符形式为 **[=%[*][width][modifiers]type=]**，具体讲解如下：

参数	描述
*	这是一个可选的星号，表示数据是从流 stream 中读取的，但是可以被忽视，即它不存储在对应的参数中。
width	这指定了在当前读取操作中读取的最大字符数。
modifiers	为对应的附加参数所指向的数据指定一个不同于整型（针对 d、i 和 n）、无符号整型（针对 o、u 和 x）或浮点型（针对 e、f 和 g）的大小： h ：短整型（针对 d、i 和 n），或无符号短整型（针对 o、u 和 x） l ：长整型（针对 d、i 和 n），或无符号长整型（针对 o、u 和 x），或双精度型（针对 e、f 和 g） L ：长双精度型（针对 e、f 和 g）
type	一个字符，指定了要被读取的数据类型以及数据读取方式。具体参见下一个表格。

scanf 类型说明符：

类型	合格的输入	参数的类型
c	单个字符：读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。	char *
d	十进制整数：数字前面的 + 或 - 号是可选的。	int *
e,E,f,g,G	浮点数：包含了一个小数点、一个可选的前置符号 + 或 -、一个可选的后置字符 e 或 E，以及一个十进制数字。两个有效的实例 -732.103 和 7.12e4	float *
o	八进制整数。	int *
s	字符串。这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。	char *
u	无符号的十进制整数。	unsigned int *
x,X	十六进制整数。	int *

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 scanf() 函数的用法。

```
#include <stdio.h>

int main()
{
   char str1[20], str2[30];

   printf("请输入用户名：");
   scanf("%s", &str1);

   printf("请输入您的网站：");
   scanf("%s", &str2);

   printf("输入的用户名：%s\n", str1);
   printf("输入的网站：%s", str2);

   return(0);
}
```

让我们编译并运行上面的程序，这将在交互模式下产生以下结果：

```
请输入用户名：admin
请输入您的网站：www.w3cschool.cc

输入的用户名：admin
输入的网站：www.w3cschool.cc
```

C 库函数 - sscanf()

描述

C 库函数 **int sscanf(const char *str, const char *format, ...)** 从字符串读取格式化输入。

声明

下面是 sscanf() 函数的声明。

```
int sscanf(const char *str, const char *format, ...)
```

参数

- **str** -- 这是 C 字符串，是函数检索数据的源。
- **format** -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 *format* 说明符。
  format 说明符形式为 **[=%[*][width][modifiers]type=]**，具体讲解如下：

参数	描述
*	这是一个可选的星号，表示数据是从流 stream 中读取的，但是可以被忽视，即它不存储在对应的参数中。
width	这指定了在当前读取操作中读取的最大字符数。
modifiers	为对应的附加参数所指向的数据指定一个不同于整型（针对 d、i 和 n）、无符号整型（针对 o、u 和 x）或浮点型（针对 e、f 和 g）的大小： h ：短整型（针对 d、i 和 n），或无符号短整型（针对 o、u 和 x） l ：长整型（针对 d、i 和 n），或无符号长整型（针对 o、u 和 x），或双精度型（针对 e、f 和 g） L ：长双精度型（针对 e、f 和 g）
type	一个字符，指定了要被读取的数据类型以及数据读取方式。具体参见下一个表格。

sscanf 类型说明符：

类型	合格的输入	参数的类型
c	单个字符：读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。	char *
d	十进制整数：数字前面的 + 或 - 号是可选的。	int *
e,E,f,g,G	浮点数：包含了一个小数点、一个可选的前置符号 + 或 -、一个可选的后置字符 e 或 E，以及一个十进制数字。两个有效的实例 -732.103 和 7.12e4	float *
o	八进制整数。	int *
s	字符串。这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。	char *
u	无符号的十进制整数。	unsigned int *
x,X	十六进制整数。	int *

- 附加参数 -- 这个函数接受一系列的指针作为附加参数，每一个指针都指向一个对象，对象类型由 format 字符串中相应的 % 标签指定，参数与 % 标签的顺序相同。

  针对检索数据的 format 字符串中的每个 format 说明符，应指定一个附加参数。如果您想要把 sscanf 操作的结果存储在一个普通的变量中，您应该在标识符前放置引用运算符（&），例如：

```
int n;
sscanf (str,"%d",&amp;n);
```

返回值

如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 sscanf() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int day, year;
   char weekday[20], month[20], dtm[100];

   strcpy( dtm, "Saturday March 25 1989" );
   sscanf( dtm, "%s %s %d  %d", weekday, month, &day, &year );

   printf("%s %d, %d = %s\n", month, day, year, weekday );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
March 25, 1989 = Saturday
```

C 库函数 - fgetc()

描述

C 库函数 **int fgetc(FILE *stream)** 从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。

声明

下面是 fgetc() 函数的声明。

```
int fgetc(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要在上面执行操作的流。

返回值

该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 fgetc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   int n = 0;

   fp = fopen("file.txt","r");
   if(fp == NULL)
   {
      perror("打开文件时发生错误");
      return(-1);
   }
   do
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }while(1);

   fclose(fp);
   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。文件将作为实例中的输入：

```
We are in 2014
```

让我们编译并运行上面的程序，这将产生以下结果：

```
We are in 2014
```

C 库函数 - fgets()

描述

C 库函数 **char *fgets(char* str, int n, FILE *stream)** 从指定的流 stream 读取一行，并把它存储在 **str** 所指向的字符串内。当读取 **(n-1)** 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。

声明

下面是 fgets() 函数的声明。

```
char *fgets(char *str, int n, FILE *stream)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了要读取的字符串。
- **n** -- 这是要读取的最大字符数（包括最后的空字符）。通常是使用以 str 传递的数组长度。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要从中读取字符的流。

返回值

如果成功，该函数返回相同的 str 参数。如果到达文件末尾或者没有读取到任何字符，str 的内容保持不变，并返回一个空指针。

如果发生错误，返回一个空指针。

实例

下面的实例演示了 fgets() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char str[60];

   /* 打开用于读取的文件 */
   fp = fopen("file.txt" , "r");
   if(fp == NULL) {
      perror("打开文件时发生错误");
      return(-1);
   }
   if( fgets (str, 60, fp)!=NULL ) {
      /* 向标准输出 stdout 写入内容 */
      puts(str);
   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。文件将作为实例中的输入：

```
We are in 2014
```

让我们编译并运行上面的程序，这将产生以下结果：

```
We are in 2014
```

C 库函数 - fputc()

描述

C 库函数 **int fputc(int char, FILE *stream)** 把参数 **char** 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。

声明

下面是 fputc() 函数的声明。

```
int fputc(int char, FILE *stream)
```

参数

- **char** -- 这是要被写入的字符。该字符以其对应的 int 值进行传递。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被写入字符的流。

返回值

如果没有发生错误，则返回被写入的字符。如果发生错误，则返回 EOF，并设置错误标识符。

实例

下面的实例演示了 fputc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int ch;

   fp = fopen("file.txt", "w+");
   for( ch = 33 ; ch <= 100; ch++ )
   {
      fputc(ch, fp);
   }
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将在当前目录中创建文件 **file.txt**，它的内容如下：

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fputs()

描述

C 库函数 **int fputs(const char *str, FILE *stream)** 把字符串写入到指定的流 stream 中，但不包括空字符。

声明

下面是 fputs() 函数的声明。

```
int fputs(const char *str, FILE *stream)
```

参数

- **str** -- 这是一个数组，包含了要写入的以空字符终止的字符序列。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被写入字符串的流。

返回值

该函数返回一个非负值，如果发生错误则返回 EOF。

实例

下面的实例演示了 fputs() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "w+");

   fputs("这是 C 语言。", fp);
   fputs("这是一种系统程序设计语言。", fp);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建文件 **file.txt**，它的内容如下：

```
这是 C 语言。这是一种系统程序设计语言。
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - getc()

描述

C 库函数 **int getc(FILE *stream)** 从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。

声明

下面是 getc() 函数的声明。

```
int getc(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要在上面执行操作的流。

返回值

该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 getc() 函数的用法。

```
#include<stdio.h>

int main()
{
   char c;

   printf("请输入字符：");
   c = getc(stdin);
   printf("输入的字符：");
   putc(c, stdout);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
请输入字符：a
输入的字符：a
```

C 库函数 - getchar()

描述

C 库函数 **int getchar(void)** 从标准输入 stdin 获取一个字符（一个无符号字符）。这等同于 **getc** 带有 stdin 作为参数。

声明

下面是 getchar() 函数的声明。

```
int getchar(void)
```

参数

- **NA**

返回值

该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 getchar() 函数的用法。

```
#include <stdio.h>

int main ()
{
   char c;

   printf("请输入字符：");
   c = getchar();

   printf("输入的字符：");
   putchar(c);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
请输入字符：a
输入的字符：a
```

C 库函数 - gets()

描述

C 库函数 **char *gets(char *str)** 从标准输入 stdin 读取一行，并把它存储在 str 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。

声明

下面是 gets() 函数的声明。

```
char *gets(char *str)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了 C 字符串。

返回值

如果成功，该函数返回 str。如果发生错误或者到达文件末尾时还未读取任何字符，则返回 NULL。

实例

下面的实例演示了 gets() 函数的用法。

```
#include <stdio.h>

int main()
{
   char str[50];

   printf("请输入一个字符串：");
   gets(str);

   printf("您输入的字符串是：%s", str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
请输入一个字符串：w3cschool.cc
您输入的字符串是：w3cschool.cc
```

C 库函数 - putc()

描述

C 库函数 **int putc(int char, FILE *stream)** 把参数 **char** 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。

声明

下面是 putc() 函数的声明。

```
int putc(int char, FILE *stream)
```

参数

- **char** -- 这是要被写入的字符。该字符以其对应的 int 值进行传递。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被写入字符的流。

返回值

该函数以无符号 char 强制转换为 int 的形式返回写入的字符，如果发生错误则返回 EOF。

实例

下面的实例演示了 putc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int ch;

   fp = fopen("file.txt", "w");
   for( ch = 33 ; ch <= 100; ch++ )
   {
      putc(ch, fp);
   }
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将在当前目录中创建文件 **file.txt**，它的内容如下：

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - putchar()

描述

C 库函数 **int putchar(int char)** 把参数 char 指定的字符（一个无符号字符）写入到标准输出 stdout 中。

声明

下面是 putchar() 函数的声明。

```
int putchar(int char)
```

参数

- **char** -- 这是要被写入的字符。该字符以其对应的 int 值进行传递。

返回值

该函数以无符号 char 强制转换为 int 的形式返回写入的字符，如果发生错误则返回 EOF。

实例

下面的实例演示了 putchar() 函数的用法。

```
#include <stdio.h>

int main ()
{
   char ch;

   for(ch = 'A' ; ch <= 'Z' ; ch++) {
      putchar(ch);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

C 库函数 - puts()

描述

C 库函数 **int puts(const char *str)** 把一个字符串写入到标准输出 stdout，直到空字符，但不包括空字符。换行符会被追加到输出中。

声明

下面是 puts() 函数的声明。

```
int puts(const char *str)
```

参数

- **str** -- 这是要被写入的 C 字符串。

返回值

如果成功，该函数返回一个非负值，如果发生错误则返回 EOF。

实例

下面的实例演示了 puts() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char str1[15];
   char str2[15];

   strcpy(str1, "w3cschool");
   strcpy(str2, "w3cbird");

   puts(str1);
   puts(str2);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
w3cschool
w3cbird
```

C 库函数 - ungetc()

描述

C 库函数 **int ungetc(int char, FILE *stream)** 把字符 **char**（一个无符号字符）推入到指定的流 **stream** 中，以便它是下一个被读取到的字符。

声明

下面是 ungetc() 函数的声明。

```
int ungetc(int char, FILE *stream)
```

参数

- **char** -- 这是要被推入的字符。该字符以其对应的 int 值进行传递。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了输入流。

返回值

如果成功，则返回被推入的字符，否则返回 EOF，且流 stream 保持不变。

实例

下面的实例演示了 ungetc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   char buffer [256];

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror("打开文件时发生错误");
      return(-1);
   }
   while(!feof(fp))
   {
      c = getc (fp);
      /* 把 ! 替换为 + */
      if( c == '!' )
      {
         ungetc ('+', fp);
      }
      else
      {
         ungetc(c, fp);
      }
      fgets(buffer, 255, fp);
      fputs(buffer, stdout);
   }
   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。文件将作为实例中的输入：

```
this is w3cschool
!c standard library
!library functions and macros
```

让我们编译并运行上面的程序，这将产生以下结果：

```
this is w3cschool
+c standard library
+library functions and macros
+library functions and macros
```

C 库函数 - perror()

描述

C 库函数 **void perror(const char *str)** 把一个描述性错误消息输出到标准错误 stderr。首先输出字符串 **str**，后跟一个冒号，然后是一个空格。

声明

下面是 perror() 函数的声明。

```
void perror(const char *str)
```

参数

- **str** -- 这是 C 字符串，包含了一个自定义消息，将显示在原本的错误消息之前。

返回值

该函数不返回任何值。

实例

下面的实例演示了 perror() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   /* 首先重命名文件 */
   rename("file.txt", "newfile.txt");

   /* 现在让我们尝试打开相同的文件 */
   fp = fopen("file.txt", "r");
   if( fp == NULL ) {
      perror("Error: ");
      return(-1);
   }
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为我们尝试打开一个不存在的文件：

```
Error: : No such file or directory
```

C 标准库 - <stdlib.h>

简介

stdlib .h 头文件定义了四个变量类型、一些宏和各种通用工具函数。

库变量

下面是头文件 stdlib.h 中定义的变量类型：

变量	描述
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。
wchar_t	这是一个宽字符常量大小的整数类型。
div_t	这是 **div** 函数返回的结构。
ldiv_t	这是 **ldiv** 函数返回的结构。

库宏

下面是头文件 stdlib.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
EXIT_FAILURE	这是 exit 函数失败时要返回的值。
EXIT_SUCCESS	这是 exit 函数成功时要返回的值。
RAND_MAX	这个宏是 rand 函数返回的最大值。
MB_CUR_MAX	这个宏表示在多字节字符集中的最大字符数，不能大于 MB_LEN_MAX。

库函数

下面是头文件 stdlib.h 中定义的函数：

函数	描述
double atof(const char *str)	把参数 *str* 所指向的字符串转换为一个浮点数（类型为 double 型）。

int atoi(const char *str)	把参数 *str* 所指向的字符串转换为一个整数（类型为 int 型）。
long int atol(const char *str)	把参数 *str* 所指向的字符串转换为一个长整数（类型为 long int 型）。
double strtod(const char *str, char **endptr)	把参数 *str* 所指向的字符串转换为一个浮点数（类型为 double 型）。
long int strtol(const char *str, char **endptr, int base)	把参数 *str* 所指向的字符串转换为一个长整数（类型为 long int 型）。
unsigned long int strtoul(const char *str, char **endptr, int base)	把参数 *str* 所指向的字符串转换为一个无符号长整数（类型为 unsigned long int 型）。
void *calloc(size_t nitems, size_t size)	分配所需的内存空间，并返回一个指向它的指针。
void free(void *ptr	释放之前调用 *calloc*、*malloc* 或 *realloc* 所分配的内存空间。
void *malloc(size_t size)	分配所需的内存空间，并返回一个指向它的指针。
void *realloc(void *ptr, size_t size)	尝试重新调整之前调用 *malloc* or *calloc* 所分配的 ptr 所指向的内存块的大小。
void abort(void)	使一个异常程序终止。
int atexit(void (*func)(void))	当程序正常终止时，调用指定的函数 **func**。
void exit(int status)	是程序正常终止。
char *getenv(const char *name)	搜索 name 所指向的环境字符串，并返回相关的值给字符串。
int system(const char *string)	由 string 指定的命令传给要被命令处理器执行的主机环境。
void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))	执行二进制搜索。
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))	数组排序。
int abs(int x)	返回 x 的绝对值。
div_t div(int numer, int denom)	分子除以分母。
long int labs(long int x)	返回 x 的绝对值。
ldiv_t ldiv(long int numer, long int denom)	分子除以分母。
int rand(void)	返回一个范围在 0 到 *RAND_MAX* 之间的伪随机数。

void srand(unsigned int seed)	该函数播种由函数 **rand** 使用的随机数发生器。
int mblen(const char *str, size_t n)	返回参数 *str* 所指向的多字节字符的长度。
size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)	把参数 *str* 所指向的多字节字符的字符串转换为参数 *pwcs* 所指向的数组。
int mbtowc(whcar_t *pwc, const char *str, size_t n)	检查参数 *str* 所指向的多字节字符。
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)	把数组 *pwcs* 中存储的编码转换为多字节字符，并把它们存储在字符串 *str* 中。
int wctomb(char *str, wchar_t wchar)	检查对应于参数 *wchar* 所给出的多字节字符的编码。

C 库函数 - atof()

描述

C 库函数 **double atof(const char *str)** 把参数 **str** 所指向的字符串转换为一个浮点数（类型为 double 型）。

声明

下面是 atof() 函数的声明。

```
double atof(const char *str)
```

参数

- **str** -- 要转换为浮点数的字符串。

返回值

函数返回转换后的双精度浮点数，如果没有执行有效的转换，则返回零（0.0）。

实例

下面的实例演示了 atof() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   float val;
   char str[20];

   strcpy(str, "98993489");
   val = atof(str);
   printf("字符串值 = %s, 浮点值 = %f\n", str, val);

   strcpy(str, "w3cschool.cc");
   val = atof(str);
   printf("字符串值 = %s, 浮点值 = %f\n", str, val);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串值 = 98993489, 浮点值 = 98993488.000000
字符串值 = w3cschool.cc, 浮点值 = 0.000000
```

C 库函数 - atoi()

描述

C 库函数 **int atoi(const char *str)** 把参数 **str** 所指向的字符串转换为一个整数（类型为 int 型）。

声明

下面是 atoi() 函数的声明。

```
int atoi(const char *str)
```

参数

- **str** -- 要转换为整数的字符串。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回零。

实例

下面的实例演示了 atoi() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   int val;
   char str[20];

   strcpy(str, "98993489");
   val = atoi(str);
   printf("字符串值 = %s, 整型值 = %d\n", str, val);

   strcpy(str, "w3cschool.cc");
   val = atoi(str);
   printf("字符串值 = %s, 整型值 = %d\n", str, val);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串值 = 98993489, 整型值 = 98993489
字符串值 = w3cschool.cc, 整型值 = 0
```

C 库函数 - atol()

描述

C 库函数 **long int atol(const char *str)** 把参数 **str** 所指向的字符串转换为一个长整数（类型为 long int 型）。

声明

下面是 atol() 函数的声明。

```
long int atol(const char *str)
```

参数

- **str** -- 要转换为长整数的字符串。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回零。

实例

下面的实例演示了 atol() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   long val;
   char str[20];

   strcpy(str, "98993489");
   val = atol(str);
   printf("字符串值 = %s, 长整型值 = %ld\n", str, val);

   strcpy(str, "w3cschool.cc");
   val = atol(str);
   printf("字符串值 = %s, 长整型值 = %ld\n", str, val);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串值 = 98993489, 长整型值 = 98993489
字符串值 = w3cschool.cc, 长整型值 = 0
```

C 库函数 - strtod()

描述

C 库函数 **double strtod(const char *str, char** endptr) 把参数 str 所指向的字符串转换为一个浮点数（类型为 **double** 型）。如果 endptr** 不为空，则指向转换中最后一个字符后的字符的指针会存储在 endptr 引用的位置。

声明

下面是 strtod() 函数的声明。

```
double strtod(const char *str, char **endptr)
```

参数

- **str** -- 要转换为双精度浮点数的字符串。
- **endptr** -- 对类型为 char 的对象的引用，其值由函数设置为 **str* 中数值后的下一个字符。

返回值

该函数返回转换后的双精度浮点数，如果没有执行有效的转换，则返回零（0.0）。

实例

下面的实例演示了 strtod() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  char str[30] = "20.30300 This is test";
   char *ptr;
   double ret;

   ret = strtod(str, &ptr);
   printf("数字（double）是 %lf\n", ret);
   printf("字符串部分是 |%s|", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
数字（double）是 20.303000
字符串部分是 | This is test|
```

C 库函数 - strtol()

描述

C 库函数 **long int strtol(const char *str, char** endptr, int base) 把参数 str 所指向的字符串根据给定的 base** 转换为一个长整数（类型为 long int 型），base 必须介于 2 和 36（包含）之间，或者是特殊值 0。

声明

下面是 strtol() 函数的声明。

```
long int strtol(const char *str, char **endptr, int base)
```

参数

- **str** -- 要转换为长整数的字符串。
- **endptr** -- 对类型为 char 的对象的引用，其值由函数设置为 **str* 中数值后的下一个字符。
- **base** -- 基数，必须介于 2 和 36（包含）之间，或者是特殊值 0。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回一个零值。

实例

下面的实例演示了 strtol() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char str[30] = "2030300 This is test";
   char *ptr;
   long ret;

   ret = strtol(str, &ptr, 10);
   printf("数字（无符号长整数）是 %ld\n", ret);
   printf("字符串部分是 |%s|", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
数字（无符号长整数）是 2030300
字符串部分是 | This is test|
```

C 库函数 - strtoul()

描述

C 库函数 **unsigned long int strtoul(const char *str, char** endptr, int base) 把参数 str 所指向的字符串根据给定的 base** 转换为一个无符号长整数（类型为 unsigned long int 型），base 必须介于 2 和 36（包含）之间，或者是特殊值 0。

声明

下面是 strtoul() 函数的声明。

```
unsigned long int strtoul(const char *str, char **endptr, int base)
```

参数

- **str** -- 要转换为无符号长整数的字符串。
- **endptr** -- 对类型为 char* 的对象的引用，其值由函数设置为 str 中数值后的下一个字符。
- **base** -- 基数，必须介于 2 和 36（包含）之间，或者是特殊值 0。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回一个零值。

实例

下面的实例演示了 strtoul() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char str[30] = "2030300 This is test";
   char *ptr;
   long ret;

   ret = strtoul(str, &ptr, 10);
   printf("数字（无符号长整数）是 %lu\n", ret);
   printf("字符串部分是 |%s|", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
数字（无符号长整数）是 2030300
字符串部分是 | This is test|
```

C 库函数 - calloc()

描述

C 库函数 **void *calloc(size_t nitems, size_t size)** 分配所需的内存空间，并返回一个指向它的指针。**malloc** 和 **calloc** 之间的不同点是，malloc 不会设置内存为零，而 calloc 会设置分配的内存为零。

声明

下面是 calloc() 函数的声明。

```
void *calloc(size_t nitems, size_t size)
```

参数

- **nitems** -- 要被分配的元素个数。
- **size** -- 元素的大小。

返回值

该函数返回一个指针，指向已分配的内存。如果请求失败，则返回 NULL。

实例

下面的实例演示了 calloc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i, n;
   int *a;

   printf("要输入的元素个数：");
   scanf("%d",&n);

   a = (int*)calloc(n, sizeof(int));
   printf("输入 %d 个数字：\n",n);
   for( i=0 ; i < n ; i++ )
   {
      scanf("%d",&a[i]);
   }

   printf("输入的数字为：");
   for( i=0 ; i < n ; i++ ) {
      printf("%d ",a[i]);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
要输入的元素个数：3
输入 3 个数字：
22
55
14
输入的数字为：22 55 14
```

C 库函数 - free()

描述

C 库函数 **void free(void *ptr)** 释放之前调用 calloc、malloc 或 realloc 所分配的内存空间。

声明

下面是 free() 函数的声明。

```
void free(void *ptr)
```

参数

- **ptr** -- 指针指向一个要释放内存的内存块，该内存块之前是通过调用 malloc、calloc 或 realloc 进行分配内存的。如果传递的参数是一个空指针，则不会执行任何动作。

返回值

该函数不返回任何值。

实例

下面的实例演示了 free() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *str;

   /* 最初的内存分配 */
   str = (char *) malloc(15);
   strcpy(str, "w3cschool");
   printf("String = %s,  Address = %u\n", str, str);

   /* 重新分配内存 */
   str = (char *) realloc(str, 25);
   strcat(str, ".cc");
   printf("String = %s,  Address = %u\n", str, str);

   /* 释放已分配的内存 */
   free(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
String = w3cschool, Address = 355090448
String = w3cschool.cc, Address = 355090448
```

C 库函数 - malloc()

描述

C 库函数 **void *malloc(size_t size)** 分配所需的内存空间，并返回一个指向它的指针。

声明

下面是 malloc() 函数的声明。

```
void *malloc(size_t size)
```

参数

- **size** -- 内存块的大小，以字节为单位。

返回值

该函数返回一个指针 ，指向已分配大小的内存。如果请求失败，则返回 NULL。

实例

下面的实例演示了 malloc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *str;

   /* 最初的内存分配 */
   str = (char *) malloc(15);
   strcpy(str, "w3cschool");
   printf("String = %s,  Address = %u\n", str, str);

   /* 重新分配内存 */
   str = (char *) realloc(str, 25);
   strcat(str, ".cc");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
String = w3cschool, Address = 355090448
String = w3cschool.cc, Address = 355090448
```

C 库函数 - realloc()

描述

C 库函数 **void *realloc(void *ptr, size_t size)** 尝试重新调整之前调用 **malloc** 或 **calloc** 所分配的 **ptr** 所指向的内存块的大小。

声明

下面是 realloc() 函数的声明。

```
void *realloc(void *ptr, size_t size)
```

参数

- **ptr** -- 指针指向一个要重新分配内存的内存块，该内存块之前是通过调用 malloc、calloc 或 realloc 进行分配内存的。如果为空指针，则会分配一个新的内存块，且函数返回一个指向它的指针。
- **size** -- 内存块的新的大小，以字节为单位。如果大小为 0，且 ptr 指向一个已存在的内存块，则 ptr 所指向的内存块会被释放，并返回一个空指针。

返回值

该函数返回一个指针 ，指向重新分配大小的内存。如果请求失败，则返回 NULL。

实例

下面的实例演示了 realloc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *str;

   /* 最初的内存分配 */
   str = (char *) malloc(15);
   strcpy(str, "w3cschool");
   printf("String = %s,  Address = %u\n", str, str);

   /* 重新分配内存 */
   str = (char *) realloc(str, 25);
   strcat(str, ".cc");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
String = w3cschool, Address = 355090448
String = w3cschool.cc, Address = 355090448
```

C 库函数 - abort()

描述

C 库函数 **void abort(void)** 中止程序执行，直接从调用的地方跳出。

声明

下面是 abort() 函数的声明。

```
void abort(void)
```

参数

- **NA**

返回值

该函数不返回任何值。

实例

下面的实例演示了 abort() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   FILE *fp;

   printf("准备打开 nofile.txt\n");
   fp = fopen( "nofile.txt","r" );
   if(fp == NULL)
   {
      printf("准备终止程序\n");
      abort();
   }
   printf("准备关闭 nofile.txt\n");
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为我们尝试打开的文件 **nofile.txt** 是不存在的：

```
准备打开 nofile.txt
准备终止程序
```

C 库函数 - atexit()

描述

C 库函数 **int atexit(void (*func)(void))** 当程序正常终止时，调用指定的函数 **func**。您可以在任何地方注册你的终止函数，但它会在程序终止的时候被调用。

声明

下面是 atexit() 函数的声明。

```
int atexit(void (*func)(void))
```

参数

- **func** -- 在程序终止时被调用的函数。

返回值

如果函数成功注册，则该函数返回零，否则返回一个非零值。

实例

下面的实例演示了 atexit() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

void functionA ()
{
   printf("这是函数A\n");
}

int main ()
{
   /* 注册终止函数 */
   atexit(functionA );

   printf("启动主程序...\n");

   printf("退出主程序...\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
启动主程序...
退出主程序...
这是函数A
```

C 库函数 - exit()

描述

C 库函数 **void exit(int status)** 立即终止调用进程。任何属于该进程的打开的文件描述符都会被关闭，该进程的子进程由进程 1 继承，初始化，且会向父进程发送一个 SIGCHLD 信号。

声明

下面是 exit() 函数的声明。

```
void exit(int status)
```

参数

- **status** -- 返回给父进程的状态值。

返回值

该函数不返回值。

实例

下面的实例演示了 exit() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   printf("程序的开头....\n");

   printf("退出程序....\n");
   exit(0);

   printf("程序的结尾....\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序的开头....
退出程序....
```

C 库函数 - getenv()

描述

C 库函数 **char *getenv(const char *name)** 搜索 name 所指向的环境字符串，并返回相关的值给字符串。

声明

下面是 getenv() 函数的声明。

```
char *getenv(const char *name)
```

参数

- **name** -- 包含被请求变量名称的 C 字符串。

返回值

该函数返回一个以 null 结尾的字符串，该字符串为被请求环境变量的值。如果该环境变量不存在，则返回 NULL。

实例

下面的实例演示了 getenv() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   printf("PATH : %s\n", getenv("PATH"));
   printf("HOME : %s\n", getenv("HOME"));
   printf("ROOT : %s\n", getenv("ROOT"));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
PATH : /sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin
HOME : /
ROOT : (null)
```

C 库函数 - system()

描述

C 库函数 **int system(const char *command)** 把 **command** 指定的命令名称或程序名称传给要被命令处理器执行的主机环境，并在命令完成后返回。

声明

下面是 system() 函数的声明。

```
int system(const char *command)
```

参数

- **command** -- 包含被请求变量名称的 C 字符串。

返回值

如果发生错误，则返回值为 -1，否则返回命令的状态。

实例

下面的实例演示了 system() 函数的用法，列出了 unix 机上当前目录下所有的文件和目录。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char command[50];

   strcpy( command, "ls -l" );
   system(command);

   return(0);
}
```

让我们编译并运行上面的程序，在 unix 机上将产生以下结果：

```
drwxr-xr-x 2 apache apache 4096 Aug 22 07:25 hsperfdata_apache
drwxr-xr-x 2 railo railo 4096 Aug 21 18:48 hsperfdata_railo
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_XXGLOBAL_1
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_asp_2
srwx---- 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp
rw------ 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp_1280495620
srwx---- 1 apache apache 0 Aug 21 18:48 mod_mono_server_global
```

下面的实例演示了 system() 函数的用法，列出了 windows 机上当前目录下所有的文件和目录。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char command[50];

   strcpy( command, "dir" );
   system(command);

   return(0);
}
```

让我们编译并运行上面的程序，在 windows 机上将产生以下结果：

```
a.txt
amit.doc
sachin
saurav
file.c
```

C 库函数 - bsearch()

描述

C 库函数 **void** ***bsearch(const void*** **key, const void** ***base, size_t nitems, size_t size, int*** **(*compar)(const void** ***, const void*** **))** 对 **nitems** 对象的数组执行二分查找，**base** 指向进行查找的数组，**key** 指向要查找的元素，**size** 指定数组中每个元素的大小。

数组的内容应根据 **compar** 所对应的比较函数升序排序。

声明

下面是 bsearch() 函数的声明。

```
void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar
```

参数

- **key** -- 指向要查找的元素的指针，类型转换为 void*。
- **base** -- 指向进行查找的数组的第一个对象的指针，类型转换为 void*。
- **nitems** -- base 所指向的数组中元素的个数。
- **size** -- 数组中每个元素的大小，以字节为单位。
- **compar** -- 用来比较两个元素的函数。

返回值

如果查找成功，该函数返回一个指向数组中匹配元素的指针，否则返回空指针。.

实例

下面的实例演示了 bsearch() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int cmpfunc(const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}

int values[] = { 5, 20, 29, 32, 63 };

int main ()
{
   int *item;
   int key = 32;

   /* 使用 bsearch() 在数组中查找值 32 */
   item = (int*) bsearch (&key, values, 5, sizeof (int), cmpfunc);
   if( item != NULL )
   {
      printf("Found item = %d\n", *item);
   }
   else
   {
      printf("Item = %d could not be found\n", *item);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Found item = 32
```

C 库函数 - qsort()

描述

C 库函数 **void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))** 对数组进行排序。

声明

下面是 qsort() 函数的声明。

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void
```

参数

- **base** -- 指向要排序的数组的第一个元素的指针。
- **nitems** -- 由 base 指向的数组中元素的个数。
- **size** -- 数组中每个元素的大小，以字节为单位。
- **compar** -- 用来比较两个元素的函数。

返回值

该函数不返回任何值。

实例

下面的实例演示了 qsort() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}

int main()
{
   int n;

   printf("排序之前的列表：\n");
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }

   qsort(values, 5, sizeof(int), cmpfunc);

   printf("\n排序之后的列表：\n");
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
排序之前的列表：
88 56 100 2 25
排序之后的列表：
2 25 56 88 100
```

C 库函数 - abs()

描述

C 库函数 **int abs(int x)** 返回 **x** 的绝对值。

声明

下面是 abs() 函数的声明。

```
int abs(int x)
```

参数

- **x** -- 完整的值。

返回值

该函数返回 x 的绝对值。

实例

下面的实例演示了 abs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   int a, b;

   a = abs(5);
   printf("a 的值 = %d\n", a);

   b = abs(-10);
   printf("b 的值 = %d\n", b);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
a 的值 = 5
b 的值 = 10
```

C 库函数 - div()

描述

C 库函数 **div_t div(int numer, int denom)** 把 **numer**（分子）除以 **denom**（分母）。

声明

下面是 div() 函数的声明。

```
div_t div(int numer, int denom)
```

参数

- **numer** -- 分子。
- **denom** -- 分母。

返回值

该函数返回定义在 <cstdlib> 中的结构中的值，该结构有两个成员，如 div*t:_int quot; int rem;*。

实例

下面的实例演示了 div() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   div_t output;

   output = div(27, 4);
   printf("(27/ 4) 的商  = %d\n", output.quot);
   printf("(27/4) 的余数 = %d\n", output.rem);

   output = div(27, 3);
   printf("(27/ 3) 的商 = %d\n", output.quot);
   printf("(27/3) 的余数 = %d\n", output.rem);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
(27/ 4) 的商 = 6
(27/4) 的余数 = 3
(27/ 3) 的商 = 9
(27/3) 的余数 = 0
```

C 库函数 - labs()

描述

C 库函数 **long int labs(long int x)** 返回 **x** 的绝对值。

声明

下面是 labs() 函数的声明。

```
long int labs(long int x)
```

参数

- **x** -- 完整的值。

返回值

该函数返回 x 的绝对值。

实例

下面的实例演示了 labs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   long int a,b;

   a = labs(65987L);
   printf("a 的值 = %ld\n", a);

   b = labs(-1005090L);
   printf("b 的值 = %ld\n", b);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
a 的值 = 65987
b 的值 = 1005090
```

C 库函数 - ldiv()

描述

C 库函数 **div_t div(long int numer, long int denom)** 把 **numer**（分子）除以 **denom**（分母）。

声明

下面是 ldiv() 函数的声明。

```
div_t div(long int numer, long int denom)
```

参数

- **numer** -- 分子。
- **denom** -- 分母。

返回值

该函数返回定义在 <cstdlib> 中的结构中的值，该结构有两个成员，如 ldiv*t:_long quot; long rem;*。

实例

下面的实例演示了 ldiv() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   ldiv_t output;

   output = ldiv(100000L, 30000L);

   printf("商 = %ld\n", output.quot);

   printf("余数 = %ld\n", output.rem);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
商 = 3
余数 = 10000
```

C 库函数 - rand()

描述

C 库函数 **int rand(void)** 返回一个范围在 0 到 *RAND_MAX* 之间的伪随机数。

RAND_MAX 是一个常量，它的默认值在不同的实现中会有所不同，但是值至少是 32767。

声明

下面是 rand() 函数的声明。

```
int rand(void)
```

参数

- **NA**

返回值

该函数返回一个范围在 0 到 RAND_MAX 之间的整数值。

实例

下面的实例演示了 rand() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i, n;
   time_t t;

   n = 5;

   /* 初始化随机数发生器 */
   srand((unsigned) time(&t));

   /* 输出 0 到 49 之间的 5 个随机数 */
   for( i = 0 ; i < n ; i++ ) {
      printf("%d\n", rand() % 50);
   }

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
38
45
29
29
47
```

C 库函数 - srand()

描述

C 库函数 **void srand(unsigned int seed)** 播种由函数 **rand** 使用的随机数发生器。

声明

下面是 srand() 函数的声明。

```
void srand(unsigned int seed)
```

参数

- **seed** -- 这是一个整型值，用于伪随机数生成算法播种。

返回值

该函数不返回任何值。

实例

下面的实例演示了 srand() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   int i, n;
   time_t t;
   
   n = 5;
   
   /* 初始化随机数发生器 */
   srand((unsigned) time(&t));

   /* 输出 0 到 50 之间的 5 个随机数 */
   for( i = 0 ; i < n ; i++ ) {
      printf("%d\n", rand() % 50);
   }
   
  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
38
45
29
29
47
```

C 库函数 - mblen()

描述

C 库函数 **int mblen(const char *str, size_t n)** 返回参数 **str** 所指向的多字节字符的长度。

声明

下面是 mblen() 函数的声明。

```
int mblen(const char *str, size_t n)
```

参数

- **str** -- 指向多字节字符的第一个字节的指针。
- **n** -- 要检查的字符长度的最大字节数。

返回值

如果识别了一个非空宽字符，mblen() 函数返回 str 开始的多字节序列解析的字节数。如果识别了一个空宽字符，则返回 0。如果识别了一个无效的多字节序列，或者不能解析一个完整的多字节字符，则返回 -1。

实例

下面的实例演示了 mblen() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   int len;
   char *pmbnull  = NULL;
   char *pmb = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwc = L"Hi";
   wchar_t *pwcs = (wchar_t *)malloc( sizeof( wchar_t ));

   printf("转换为多字节字符串\n");
   len = wcstombs( pmb, pwc, MB_CUR_MAX);
   printf("被转换的字符 %d\n", len);
   printf("第一个多字节字符的十六进制值：%#.4x\n", pmb);

   len = mblen( pmb, MB_CUR_MAX );
   printf( "多字节字符 %x 的字节长度：%u\n", pmb, len );

   pmb = NULL;

   len = mblen( pmb, MB_CUR_MAX );
   printf( "多字节字符 %x 的字节长度：%u\n", pmb, len );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
转换为多字节字符串
被转换的字符 1
第一个多字节字符的十六进制值：0x168c6010
多字节字符 168c6010 的字节长度：1
多字节字符 0 的字节长度：0
```

C 库函数 - mbstowcs()

描述

C 库函数 **size_t mbstowcs(schar_t *pwcs, const char* str, size_t n)** 把参数 **str** 所指向的多字节字符的字符串转换为参数 **pwcs** 所指向的数组。

声明

下面是 mbstowcs() 函数的声明。

```
size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)
```

参数

- **pwcs** -- 指向一个 wchar_t 元素的数组，数组长度足以存储一个最大字符长度的宽字符串。
- **str** -- 要被转换的多字节字符字符串。
- **n** -- 要被转换的最大字符数。

返回值

该函数返回转换的字符数，不包括结尾的空字符。如果遇到一个无效的多字节字符，则返回 -1 值。

实例

下面的实例演示了 mbstowcs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   int len;
   char *pmbnull  = NULL;
   char *pmb = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwc = L"Hi";
   wchar_t *pwcs = (wchar_t *)malloc( sizeof( wchar_t ));

   printf("转换为多字节字符串\n");
   len = wcstombs( pmb, pwc, MB_CUR_MAX);
   printf("被转换的字符 %d\n", len);
   printf("第一个多字节字符的十六进制值：%#.4x\n", pmb);

   printf("转换回宽字符字符串\n");
   len = mbstowcs( pwcs, pmb, MB_CUR_MAX);
   printf("被转换的字符 %d\n", len);
   printf("第一个宽字符的十六进制值：%#.4x\n\n", pwcs);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
转换为多字节字符串
被转换的字符 1
第一个多字节字符的十六进制值：0x19a60010
转换回宽字符字符串
被转换的字符 1
第一个宽字符的十六进制值：0x19a60030
```

C 库函数 - mbtowc()

描述

C 库函数 **int mbtowc(whcar_t *pwc, const char *str, size_t n)** 把一个多字节序列转换为一个宽字符。

声明

下面是 mbtowc() 函数的声明。

```
int mbtowc(whcar_t *pwc, const char *str, size_t n)
```

参数

- **pwc** -- 指向类型为 wchar_t 对象的指针。
- **str** -- 指向多字节字符的第一个字节的指针。
- **n** -- 要被检查的最大字节数。

返回值

- 如果 str 不为 NULL，mbtowc() 函数返回 str 开始消耗的字节数，如果指向一个空字节，则返回 0，如果操作失败，则返回 -1。
- 如果 str 为 NULL，如果编码具有移位状态，则 mbtowc() 函数返回非零，如果编码是无状态的，则返回零。

实例

下面的实例演示了 mbtowc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   char *str = "这里是 w3cschool.cc";
   wchar_t mb[100];
   int len;

   len = mblen(NULL, MB_CUR_MAX);

   mbtowc(mb, str, len*strlen(str) );

   wprintf(L"%ls \n", mb );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为它要以多字节形式输出结果，这是一种二进制输出。

```
???
```

C 库函数 - wcstombs()

描述

C 库函数 **size_t wcstombs(char *str, const wchar_t* **pwcs, size_t n)** 把宽字符字符串 **pwcs** 转换为一个 **str** 开始的多字节字符串。最多会有 **n** 个字节被写入 **str** 中。

声明

下面是 wcstombs() 函数的声明。

```
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)
```

参数

- **str** -- 指向一个 char 元素的数组，至少有 n 字节长。
- **pwcs** -- 要被转换的宽字符字符串。
- **n** -- 要被写入到 str 中的最大字节数。

返回值

该函数返回转换和写入到 str 中的字节数，不包括结尾的空字符。如果遇到一个无效的多字节字符，则返回 -1 值。

实例

下面的实例演示了 wcstombs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 50

int main()
{
   size_t ret;
   char *MB = (char *)malloc( BUFFER_SIZE );
   wchar_t *WC = L"http://www.w3cschool.cc";

   /* 转换宽字符字符串 */
   ret = wcstombs(MB, WC, BUFFER_SIZE);

   printf("要转换的字符数 = %u\n", ret);
   printf("多字节字符 = %s\n\n", MB);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
要转换的字符数 = 23
多字节字符 = http://www.w3cschool.cc
```

C 库函数 - wctomb()

描述

C 库函数 **int wctomb(char *str, wchar_t wchar)** 把宽字符 **wchar** 转换为它的多字节表示形式，并把它存储在 **str** 指向的字符数组的开头。

声明

下面是 wctomb() 函数的声明。

```
int wctomb(char *str, wchar_t wchar)
```

参数

- **str** -- 一个指针，指向一个足以存储多字节字符的数组。
- **wchar** -- 类型为 wchar_t 的宽字符。

返回值

- 如果 str 不为 NULL，wctomb() 函数返回写入字节数组中的字节数。如果 wchar 不能被表示为一个多字节序列，则会返回 -1。
- 如果 str 为 NULL，如果编码具有移位状态，则 wctomb() 函数返回非零，如果编码是无状态的，则返回零。

实例

下面的实例演示了 wctomb() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i;
   wchar_t wc = L'a';
   char *pmbnull = NULL;
   char *pmb = (char *)malloc(sizeof( char ));

   printf("要转换的宽字符：\n");
   i = wctomb( pmb, wc );
   printf("被转换的字符：%u\n", i);
   printf("多字节字符：%.1s\n", pmb);

   printf("当要转换的字符为 NULL 时尝试转换：\n");
   i = wctomb( pmbnull, wc );
   printf("被转换的字符：%u\n", i);
   /* 不会输出任何值 */
   printf("多字节字符：%.1s\n", pmbnull);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
要转换的宽字符：
被转换的字符：1
多字节字符：a
当要转换的字符为 NULL 时尝试转换：
被转换的字符：0
多字节字符：
```

C 标准库 - <string.h>

简介

string .h 头文件定义了一个变量类型、一个宏和各种操作字符数组的函数。

库变量

下面是头文件 string.h 中定义的变量类型：

变量	描述
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。

库宏

下面是头文件 string.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。

库函数

下面是头文件 string.h 中定义的函数：

函数	描述
void *memchr(const void *str, int c, size_t n)	在参数 *str* 所指向的字符串的前 n 个字节中搜索第一次出现字符 c（一个无符号字符）的位置。
int memcmp(const void *str1, const void *str2, size_t n)	把 *str1* 和 *str2* 的前 n 个字节进行比较。
void *memcpy(void *dest, const void *src, size_t n)	从 src 复制 n 个字符到 *dest*。
void	

*memmove(void *dest, const void *src, size_t n)	另一个用于从 *str2* 复制 n 个字符到 *str1* 的函数。
void *memset(void *str, int c, size_t n)	复制字符 c（一个无符号字符）到参数 *str* 所指向的字符串的前 n 个字符。
char *strcat(char *dest, const char *src)	把 *src* 所指向的字符串追加到 *dest* 所指向的字符串的结尾。
char *strncat(char *dest, const char *src, size_t n)	把 *src* 所指向的字符串追加到 *dest* 所指向的字符串的结尾，直到 n 字符长度为止。
char *strchr(const char *str, int c)	在参数 *str* 所指向的字符串中搜索第一次出现字符 c（一个无符号字符）的位置。
int strcmp(const char *str1, const char *str2)	把 *str1* 所指向的字符串和 *str2* 所指向的字符串进行比较。
int strncmp(const char *str1, const char *str2, size_t n)	把 *str1* 和 *str2* 进行比较，最多比较前 n 个字节。
int strcoll(const char *str1, const char *str2)	把 *str1* 和 *str2* 进行比较，结果取决于 LC_COLLATE 的位置设置。
char *strcpy(char *dest, const char *src)	把 *src* 所指向的字符串复制到 *dest*。
char *strncpy(char *dest, const char *src, size_t n)	把 *src* 所指向的字符串复制到 *dest*，最多复制 n 个字符。
size_t strcspn(const char *str1, const char *str2)	检索字符串 str1 开头连续有几个字符都不含字符串 str2 中的字符。
char *strerror(int errnum)	从内部数组中搜索错误号 errnum，并返回一个指向错误消息字符串的指针。
size_t strlen(const char *str)	计算字符串 str 的长度，直到空结束字符串，但不包括空结束字符串。
char *strpbrk(const char *str1, const char *str2)	检索字符串 *str1* 中匹配字符串 *str2* 中所指定的字符的第一个字符。也就是说，依次检验字符串s1中的字符，当被检验字符在字符串s2中也包含时，则停止检验，并返回该字符位置。
char *strrchr(const	在参数 *str* 所指向的字符串中搜索最后一次出现字符 c（一个无符

char *str, int c)	号字符）的位置。
size_t strspn(const char *str1, const char *str2)	检索字符串 *str1* 开头连续有几个字符都不含字符串 *str2* 中的字符。
char *strstr(const char *haystack, const char *needle)	在字符串 *haystack* 中查找第一次出现字符串 *needle*（不包含空结束字符串）的位置。
char *strtok(char *str, const char *delim)	分解字符串 *str* 为一组字符串，*delim* 为分隔符。
size_t strxfrm(char *dest, const char *src, size_t n)	根据程序当前的区域选项中的 LC_COLLATE 来转换字符串 **src** 的前 **n** 个字符，并把它们放置在字符串 **dest** 中。

C 库函数 - memchr()

描述

C 库函数 **void *memchr(const void* str, int c, size_t n)** 在参数 **str** 所指向的字符串的前 **n** 个字节中搜索第一次出现字符 **c**（一个无符号字符）的位置。

声明

下面是 memchr() 函数的声明。

```
void *memchr(const void *str, int c, size_t n)
```

参数

- **str** -- 指向要执行搜索的内存块。
- **c** -- 以 int 形式传递的值，但是函数在每次字节搜索时是使用该值的无符号字符形式。
- **n** -- 要被分析的字节数。

返回值

该函数返回一个指向匹配字节的指针，如果在给定的内存区域未出现字符，则返回 NULL。

实例

下面的实例演示了 memchr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char str[] = "http://www.w3cschool.cc";
   const char ch = '.';
   char *ret;

   ret = memchr(str, ch, strlen(str));

   printf("|%c| 之后的字符串是 - |%s|\n", ch, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|.| 之后的字符串是 - |.w3cschool.cc|
```

C 库函数 - memcmp()

描述

C 库函数 **int memcmp(const void *str1, const void* str2, size_t n))** 把存储区 **str1** 和存储区 **str2** 的前 **n** 个字节进行比较。

声明

下面是 memcmp() 函数的声明。

```
int memcmp(const void *str1, const void *str2, size_t n)
```

参数

- **str1** -- 指向内存块的指针。
- **str2** -- 指向内存块的指针。
- **n** -- 要被比较的字节数。

返回值

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 memcmp() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   memcpy(str1, "abcdef", 6);
   memcpy(str2, "ABCDEF", 6);

   ret = memcmp(str1, str2, 5);

   if(ret > 0)
   {
      printf("str2 小于 str1");
   }
   else if(ret > 0)
   {
      printf("str1 小于 str2");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str2 小于 str1
```

C 库函数 - memcpy()

描述

C 库函数 **void *memcpy(void *str1, const void *str2, size_t n)** 从存储区 **str2** 复制 **n** 个字符到存储区 **str1**。

声明

下面是 memcpy() 函数的声明。

```
void *memcpy(void *str1, const void *str2, size_t n)
```

参数

- **str1** -- 指向用于存储复制内容的目标数组，类型强制转换为 void* 指针。
- **str2** -- 指向要复制的数据源，类型强制转换为 void* 指针。
- **n** -- 要被复制的字节数。

返回值

该函数返回一个指向目标存储区 str1 的指针。

实例

下面的实例演示了 memcpy() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char src[50] = "http://www.w3cschool.cc";
   char dest[50];

   printf("Before memcpy dest = %s\n", dest);
   memcpy(dest, src, strlen(src)+1);
   printf("After memcpy dest = %s\n", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Before memcpy dest =
After memcpy dest = http://www.w3cschool.cc
```

C 库函数 - memmove()

描述

C 库函数 **void *memmove(void* str1, const void *str2, size_t n)** 从 **str2** 复制 **n** 个字符到 **str1**，但是在重叠内存块这方面，memmove() 是比 memcpy() 更安全的方法。如果目标区域和源区域有重叠的话，memmove() 能够保证源串在被覆盖之前将重叠区域的字节拷贝到目标区域中，复制后源区域的内容会被更改。如果目标区域与源区域没有重叠，则和 memcpy() 函数功能相同。

声明

下面是 memmove() 函数的声明。

```
void *memmove(void *str1, const void *str2, size_t n)
```

参数

- **str1** -- 指向用于存储复制内容的目标数组，类型强制转换为 void* 指针。
- **str2** -- 指向要复制的数据源，类型强制转换为 void* 指针。
- **n** -- 要被复制的字节数。

返回值

该函数返回一个指向目标存储区 str1 的指针。

实例

下面的实例演示了 memmove() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char dest[] = "oldstring";
   const char src[]  = "newstring";

   printf("Before memmove dest = %s, src = %s\n", dest, src);
   memmove(dest, src, 9);
   printf("After memmove dest = %s, src = %s\n", dest, src);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Before memmove dest = oldstring, src = newstring
After memmove dest = newstring, src = newstring
```

C 库函数 - memset()

描述

C 库函数 **void *memset(void* str, int c, size_t n)** 复制字符 **c**（一个无符号字符）到参数 **str** 所指向的字符串的前 **n** 个字符。

声明

下面是 memset() 函数的声明。

```
void *memset(void *str, int c, size_t n)
```

参数

- **str** -- 指向要填充的内存块。
- **c** -- 要被设置的值。该值以 int 形式传递，但是函数在填充内存块时是使用该值的无符号字符形式。
- **n** -- 要被设置为该值的字节数。

返回值

该值返回一个指向存储区 str 的指针。

实例

下面的实例演示了 memset() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str[50];

   strcpy(str,"This is string.h library function");
   puts(str);

   memset(str,'$',7);
   puts(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
This is string.h library function
$$$$$$$ string.h library function
```

C 库函数 - strcat()

描述

C 库函数 **char** ***strcat(char*** **dest, const char *src)** 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾。

声明

下面是 strcat() 函数的声明。

```
char *strcat(char *dest, const char *src)
```

参数

- **dest** -- 指向目标数组，该数组包含了一个 C 字符串，且足够容纳追加后的字符串。
- **src** -- 指向要追加的字符串，该字符串不会覆盖目标字符串。

返回值

该函数返回一个指向最终的目标字符串 dest 的指针。

实例

下面的实例演示了 strcat() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char src[50], dest[50];

   strcpy(src,  "This is source");
   strcpy(dest, "This is destination");

   strcat(dest, src);

   printf("最终的目标字符串： |%s|", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： |This is destinationThis is source|
```

C 库函数 - strncat()

描述

C 库函数 **char** ***strncat(char*** **dest, const char *src, size_t n)** 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾，直到 **n** 字符长度为止。

声明

下面是 strncat() 函数的声明。

```
char *strncat(char *dest, const char *src, size_t n)
```

参数

- **dest** -- 指向目标数组，该数组包含了一个 C 字符串，且足够容纳追加后的字符串，包括额外的空字符。
- **src** -- 要追加的字符串。
- **n** -- 要追加的最大字符数。

返回值

该函数返回一个指向最终的目标字符串 dest 的指针。

实例

下面的实例演示了 strncat() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char src[50], dest[50];

   strcpy(src,  "This is source");
   strcpy(dest, "This is destination");

   strncat(dest, src, 15);

   printf("最终的目标字符串： |%s|", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： |This is destinationThis is source|
```

C 库函数 - strchr()

描述

C 库函数 **char *strchr(const char* str, int c)** 在参数 **str** 所指向的字符串中搜索第一次出现字符 **c**（一个无符号字符）的位置。

声明

下面是 strchr() 函数的声明。

```
char *strchr(const char *str, int c)
```

参数

- **str** -- 要被检索的 C 字符串。
- **c** -- 在 str 中要搜索的字符。

返回值

该函数返回在字符串 str 中第一次出现字符 c 的位置，如果未找到该字符则返回 NULL。

实例

下面的实例演示了 strchr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char str[] = "http://www.w3cschool.cc";
   const char ch = '.';
   char *ret;

   ret = strchr(str, ch);

   printf("|%c| 之后的字符串是 - |%s|\n", ch, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|.| 之后的字符串是 - |.w3cschool.cc|
```

C 库函数 - strcmp()

描述

C 库函数 **int strcmp(const char *str1, const char *str2)** 把 **str1** 所指向的字符串和 **str2** 所指向的字符串进行比较。

声明

下面是 strcmp() 函数的声明。

```
int strcmp(const char *str1, const char *str2)
```

参数

- **str1** -- 要进行比较的第一个字符串。
- **str2** -- 要进行比较的第二个字符串。

返回值

该函数返回值如下：

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 strncmp() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strcmp(str1, str2);

   if(ret < 0)
   {
      printf("str1 小于 str2");
   }
   else if(ret > 0)
   {
      printf("str2 小于 str1");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str2 小于 str1
```

C 库函数 - strncmp()

描述

C 库函数 **int strncmp(const char *str1, const char* str2, size_t n)** 把 **str1** 和 **str2** 进行比较，最多比较前 **n** 个字节。

声明

下面是 strncmp() 函数的声明。

```
int strncmp(const char *str1, const char *str2, size_t n)
```

参数

- **str1** -- 要进行比较的第一个字符串。
- **str2** -- 要进行比较的第二个字符串。
- **n** -- 要比较的最大字符数。

返回值

该函数返回值如下：

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 strncmp() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strncmp(str1, str2, 4);

   if(ret < 0)
   {
      printf("str1 小于 str2");
   }
   else if(ret > 0)
   {
      printf("str2 小于 str1");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str2 小于 str1
```

C 库函数 - strcoll()

描述

C 库函数 **int strcoll(const char *str1, const char *str2)** 把 **str1** 和 **str2** 进行比较，结果取决于 LC_COLLATE 的位置设置。

声明

下面是 strcoll() 函数的声明。

```
int strcoll(const char *str1, const char *str2)
```

参数

- **str1** -- 要进行比较的第一个字符串。
- **str2** -- 要进行比较的第二个字符串。

返回值

该函数返回值如下：

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 strcoll() 函数的用法。.

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   strcpy(str1, "abc");
   strcpy(str2, "ABC");

   ret = strcoll(str1, str2);

   if(ret > 0)
   {
      printf("str1 小于 str2");
   }
   else if(ret < 0)
   {
      printf("str2 小于 str1");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str1 小于 str2
```

C 库函数 - strcpy()

描述

C 库函数 **char *strcpy(char *dest, const char *src)** 把 **src** 所指向的字符串复制到 **dest**。

声明

下面是 strcpy() 函数的声明。

```
char *strcpy(char *dest, const char *src)
```

参数

- **dest** -- 指向用于存储复制内容的目标数组。
- **src** -- 要复制的字符串。

返回值

该函数返回一个指向最终的目标字符串 dest 的指针。

实例

下面的实例演示了 strcpy() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char src[40];
   char dest[100];

   memset(dest, '\0', sizeof(dest));
   strcpy(src, "This is w3cschool.cc");
   strcpy(dest, src);

   printf("最终的目标字符串： %s\n", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： This is w3cschool.cc
```

C 库函数 - strncpy()

描述

C 库函数 **char *strncpy(char *dest, const char *src, size_t n)** 把 **src** 所指向的字符串复制到 **dest**，最多复制 **n** 个字符。当 src 的长度小于 n 时，dest 的剩余部分将用空字节填充。

声明

下面是 strncpy() 函数的声明。

```
char *strncpy(char *dest, const char *src, size_t n)
```

参数

- **dest** -- 指向用于存储复制内容的目标数组。
- **src** -- 要复制的字符串。
- **n** -- 要从源中复制的字符数。

返回值

该函数返回最终复制的字符串。

实例

下面的实例演示了 strncpy() 函数的用法。在这里，我们使用函数 memset() 来清除内存位置。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char src[40];
   char dest[12];

   memset(dest, '\0', sizeof(dest));
   strcpy(src, "This is w3cschool.cc");
   strncpy(dest, src, 10);

   printf("最终的目标字符串： %s\n", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： This is w3
```

C 库函数 - strcspn()

描述

C 库函数 **size_t strcspn(const char *str1, const char *str2)** 检索字符串 **str1** 开头连续有几个字符都不含字符串 **str2** 中的字符。

声明

下面是 strcspn() 函数的声明。

```
size_t strcspn(const char *str1, const char *str2)
```

参数

- **str1** -- 要被检索的 C 字符串。
- **str2** -- 该字符串包含了要在 str1 中进行匹配的字符列表。

返回值

该函数返回 str1 开头连续都不含字符串 str2 中字符的字符数。

实例

下面的实例演示了 strcspn() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int len;
   const char str1[] = "ABCDEF4960910";
   const char str2[] = "013";

   len = strcspn(str1, str2);

   printf("第一个匹配的字符是在 %d\n", len + 1);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
第一个匹配的字符是在 10
```

C 库函数 - strerror()

描述

C 库函数 **char *strerror(int errnum)** 从内部数组中搜索错误号 **errnum**，并返回一个指向错误消息字符串的指针。**strerror** 生成的错误字符串取决于开发平台和编译器。

声明

下面是 strerror() 函数的声明。

```
char *strerror(int errnum)
```

参数

- **errnum** -- 错误号，通常是 **errno**。

返回值

该函数返回一个指向错误字符串的指针，该错误字符串描述了错误 errnum。

实例

下面的实例演示了 strerror() 函数的用法。

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","r");
   if( fp == NULL )
   {
      printf("Error: %s\n", strerror(errno));
   }

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为我们尝试打开一个不存在的文件：

```
Error: No such file or directory
```

C 库函数 - strlen()

描述

C 库函数 **size_t strlen(const char *str)** 计算字符串 **str** 的长度，直到空结束字符，但不包括空结束字符。

声明

下面是 strlen() 函数的声明。

```
size_t strlen(const char *str)
```

参数

- **str** -- 要计算长度的字符串。

返回值

该函数返回字符串的长度。

实例

下面的实例演示了 strlen() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str[50];
   int len;

   strcpy(str, "This is w3cschool.cc");

   len = strlen(str);
   printf("|%s| 的长度是 |%d|\n", str, len);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|This is w3cschool.cc| 的长度是 |20|
```

C 库函数 - strpbrk()

描述

C 库函数 **char *strpbrk(const char* str1, const char *str2)** 检索字符串 **str1** 中第一个匹配字符串 **str2** 中字符的字符，不包含空结束字符。也就是说，依次检验字符串 str1 中的字符，当被检验字符在字符串 str2 中也包含时，则停止检验，并返回该字符位置。

声明

下面是 strpbrk() 函数的声明。

```
char *strpbrk(const char *str1, const char *str2)
```

参数

- **str1** -- 要被检索的 C 字符串。
- **str2** -- 该字符串包含了要在 str1 中进行匹配的字符列表。

返回值

该函数返回 str1 中第一个匹配字符串 str2 中字符的字符数，如果未找到字符则返回 NULL。

实例

下面的实例演示了 strpbrk() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char str1[] = "abcde2fghi3jk4l";
   const char str2[] = "34";
   char *ret;

   ret = strpbrk(str1, str2);
   if(ret)
   {
      printf("第一个匹配的字符是： %c\n", *ret);
   }
   else
   {
      printf("未找到字符");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
第一个匹配的字符是： 3
```

C 库函数 - strrchr()

描述

C 库函数 **char *strrchr(const char* str, int c)** 在参数 **str** 所指向的字符串中搜索最后一次出现字符 **c**（一个无符号字符）的位置。

声明

下面是 strrchr() 函数的声明。

```
char *strrchr(const char *str, int c)
```

参数

- **str** -- C 字符串。
- **c** -- 要搜索的字符。以 int 形式传递，但是最终会转换回 char 形式。

返回值

该函数返回 str 中最后一次出现字符 c 的位置。如果未找到该值，则函数返回一个空指针。

实例

下面的实例演示了 strrchr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int len;
   const char str[] = "http://www.w3cschool.cc";
   const char ch = '.';
   char *ret;

   ret = strrchr(str, ch);

   printf("|%c| 之后的字符串是 - |%s|\n", ch, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|.| 之后的字符串是 - |.cc|
```

C 库函数 - strspn()

描述

C 库函数 **size_t strspn(const char *str1, const char* str2)** 检索字符串 **str1** 中第一个不在字符串 **str2** 中出现的字符下标。

声明

下面是 strspn() 函数的声明。

```
size_t strspn(const char *str1, const char *str2)
```

参数

- **str1** -- 要被检索的 C 字符串。
- **str2** -- 该字符串包含了要在 str1 中进行匹配的字符列表。

返回值

该函数返回 str1 中第一个不在字符串 str2 中出现的字符下标。

实例

下面的实例演示了 strspn() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int len;
   const char str1[] = "ABCDEFG019874";
   const char str2[] = "ABCD";

   len = strspn(str1, str2);

   printf("初始段匹配长度 %d\n", len );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
初始段匹配长度 4
```

C 库函数 - strstr()

描述

C 库函数 **char *strstr(const char* haystack, const char *needle)** 在字符串 **haystack** 中查找第一次出现字符串 **needle** 的位置，不包含终止符 '\0'。

声明

下面是 strstr() 函数的声明。

```
char *strstr(const char *haystack, const char *needle)
```

参数

- **haystack** -- 要被检索的 C 字符串。
- **needle** -- 在 haystack 字符串内要搜索的小字符串。

返回值

该函数返回在 haystack 中第一次出现 needle 字符串的位置，如果未找到则返回 null。

实例

下面的实例演示了 strstr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   const char haystack[20] = "W3CSchool";
   const char needle[10] = "School";
   char *ret;

   ret = strstr(haystack, needle);

   printf("子字符串是： %s\n", ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
子字符串是： School
```

C 库函数 - strtok()

描述

C 库函数 **char *strtok(char* **str, const char *delim)** 分解字符串 **str** 为一组字符串，**delim** 为分隔符。

声明

下面是 strtok() 函数的声明。

```
char *strtok(char *str, const char *delim)
```

参数

- **str** -- 要被分解成一组小字符串的字符串。
- **delim** -- 包含分隔符的 C 字符串。

返回值

该函数返回被分解的最后一个子字符串，如果没有可检索的字符串，则返回一个空指针。

实例

下面的实例演示了 strtok() 函数的用法。

```
#include <string.h>
#include <stdio.h>

int main()
{
   const char str[80] = "This is - www.w3cschool.cc - website";
   const char s[2] = "-";
   char *token;

   /* 获取第一个子字符串 */
   token = strtok(str, s);

   /* 继续获取其他的子字符串 */
   while( token != NULL )
   {
      printf( " %s\n", token );

      token = strtok(NULL, s);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
This is
www.w3cschool.cc
website
```

C 库函数 - strxfrm()

描述

C 库函数 **size_t strxfrm(char *dest, const char* src, size_t n)** 根据程序当前的区域选项中的 LC_COLLATE 来转换字符串 **src** 的前 **n** 个字符，并把它们放置在字符串 **dest** 中。

声明

下面是 strxfrm() 函数的声明。

```
size_t strxfrm(char *dest, const char *src, size_t n)
```

参数

- **dest** -- 指向存储内容的目标数组的指针，如果参数 n 为 0，则它是一个空指针。
- **src** -- 要被转换为当前区域设置的 C 字符串。
- **n** -- 被复制到 str1 的最大字符数。

返回值

该函数返回被转换字符串的长度，不包括空结束字符。

实例

下面的实例演示了 strxfrm() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char dest[20];
   char src[20];
   int len;

   strcpy(src, "W3C School");
   len = strxfrm(dest, src, 20);

   printf("字符串 |%s| 的长度是： |%d|", dest, len);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串 |W3C School| 的长度是： |10|
```

C 标准库 - <time.h>

简介

time.h 头文件定义了四个变量类型、两个宏和各种操作日期和时间的函数。

库变量

下面是头文件 time.h 中定义的变量类型：

变量	描述
size_t	是无符号整数类型，它是 **sizeof** 关键字的结果。
clock_t	这是一个适合存储处理器时间的类型。
time_t is	这是一个适合存储日历时间类型。
struct tm	这是一个用来保存时间和日期的结构。

tm 结构的定义如下：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59        */
   int tm_min;         /* 分，范围从 0 到 59        */
   int tm_hour;        /* 小时，范围从 0 到 23        */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31    */
   int tm_mon;         /* 月，范围从 0 到 11        */
   int tm_year;        /* 自 1900 年起的年数        */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6    */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365    */
   int tm_isdst;       /* 夏令时                */
};
```

库宏

下面是头文件 time.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
CLOCKS_PER_SEC	这个宏表示每秒的处理器时钟个数。

库函数

下面是头文件 time.h 中定义的函数：

函数	描述
char *asctime(const struct tm *timeptr)	返回一个指向字符串的指针，它代表了结构 timeptr 的日期和时间。
clock_t clock(void)	返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。
char *ctime(const time_t *timer)	返回一个表示当地时间的字符串，当地时间是基于参数 timer。
double difftime(time_t time1, time_t time2)	返回 time1 和 time2 之间相差的秒数 (time1-time2)。
struct tm *gmtime(const time_t *timer)	timer 的值被分解为tm 结构，并用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。
struct tm *localtime(const time_t *timer)	timer 的值被分解为tm 结构，并用本地时区表示。
time_t mktime(struct tm *timeptr)	把 timeptr 所指向的结构转换为一个依据本地时区的 time_t 值。
size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)	根据 format 中定义的格式化规则，格式化结构 timeptr 表示的时间，并把它存储在 str 中。
time_t time(time_t *timer)	计算当前日历时间，并把它编码成 time_t 格式。

C 库函数 - asctime()

描述

C 库函数 **char *asctime(const struct tm* timeptr)** 返回一个指向字符串的指针，它代表了结构 **struct timeptr** 的日期和时间。

声明

下面是 asctime() 函数的声明。

```
char *asctime(const struct tm *timeptr)
```

参数

timeptr 是指向 tm 结构的指针，包含了分解为如下各部分的日历时间：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

返回值

该函数返回一个 C 字符串，包含了可读格式的日期和时间信息 **Www Mmm dd hh:mm:ss yyyy**，其中，*Www* 表示星期几，*Mmm* 是以字母表示的月份，*dd* 表示一月中的第几天，*hh:mm:ss* 表示时间，*yyyy* 表示年份。

实例

下面的实例演示了 asctime() 函数的用法。

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main()
{
   struct tm t;

   t.tm_sec    = 10;
   t.tm_min    = 10;
   t.tm_hour   = 6;
   t.tm_mday   = 25;
   t.tm_mon    = 2;
   t.tm_year   = 89;
   t.tm_wday   = 6;

   puts(asctime(&t));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Sat Mar 25 06:10:10 1989
```

C 库函数 - clock()

描述

C 库函数 **clock_t clock(void)** 返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。为了获取 CPU 所使用的秒数，您需要除以 CLOCKS_PER_SEC。

在 32 位系统中，CLOCKS_PER_SEC 等于 1000000，该函数大约每 72 分钟会返回相同的值。

声明

下面是 clock() 函数的声明。

```
clock_t clock(void)
```

参数

- **NA**

返回值

该函数返回自程序启动起，处理器时钟所使用的时间。如果失败，则返回 -1 值。

实例

下面的实例演示了 clock() 函数的用法。

```
#include <time.h>
#include <stdio.h>

int main()
{
   clock_t start_t, end_t, total_t;
   int i;

   start_t = clock();
   printf("程序启动，start_t = %ld\n", start_t);

   printf("开始一个大循环，start_t = %ld\n", start_t);
   for(i=0; i< 10000000; i++)
   {
   }
   end_t = clock();
   printf("大循环结束，end_t = %ld\n", end_t);

   total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
   printf("CPU 占用的总时间：%f\n", total_t  );
   printf("程序退出...\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序启动，start_t = 0
开始一个大循环，start_t = 0
大循环结束，end_t = 20000
CPU 占用的总时间：0.000000
程序退出...
```

C 库函数 - ctime()

描述

C 库函数 **char *ctime(const time_t* timer)** 返回一个表示当地时间的字符串，当地时间是基于参数 **timer**。

返回的字符串格式如下： **Www Mmm dd hh:mm:ss yyyy** 其中，*Www* 表示星期几，*Mmm* 是以字母表示的月份，*dd* 表示一月中的第几天，*hh:mm:ss* 表示时间，*yyyy* 表示年份。

声明

下面是 ctime() 函数的声明。

```
char *ctime(const time_t *timer)
```

参数

- **timer** -- 这是指向 time_t 对象的指针，该对象包含了一个日历时间。

返回值

该函数返回一个 C 字符串，该字符串包含了可读格式的日期和时间信息。

实例

下面的实例演示了 ctime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t curtime;

   time(&curtime);

   printf("当前时间 = %s", ctime(&curtime));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前时间 = Mon Aug 13 08:23:14 2012
```

C 库函数 - difftime()

描述

C 库函数 **double difftime(time_t time1, time_t time2)** 返回 **time1** 和 **time2** 之间相差的秒数 **(time1 - time2)**。这两个时间是在日历时间中指定的，表示了自纪元 Epoch（协调世界时 UTC：1970-01-01 00:00:00）起经过的时间。

声明

下面是 difftime() 函数的声明。

```
double difftime(time_t time1, time_t time2)
```

参数

- **time1** -- 这是表示结束时间的 time_t 对象。
- **time2** -- 这是表示开始时间的 time_t 对象。

返回值

该函数返回以双精度浮点型 double 值表示的两个时间之间相差的秒数 (time2 - time1)。

实例

下面的实例演示了 difftime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t start_t, end_t;
   double diff_t;

   printf("程序启动...\n");
   time(&start_t);

   printf("休眠 5 秒...\n");
   sleep(5);

   time(&end_t);
   diff_t = difftime(end_t, start_t);

   printf("执行时间 = %f\n", diff_t);
   printf("程序退出...\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序启动...
休眠 5 秒...
执行时间 = 5.000000
程序退出...
```

C 库函数 - gmtime()

描述

C 库函数 **struct tm *gmtime(const time_t *timer)** 使用 **timer** 的值来填充 **tm** 结构，并用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。

声明

下面是 gmtime() 函数的声明。

```
struct tm *gmtime(const time_t *timer)
```

参数

- **timeptr** -- 这是指向表示日历时间的 time_t 值的指针。

返回值

该函数返回指向 tm 结构的指针，该结构带有被填充的时间信息。下面是 timeptr 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

实例

下面的实例演示了 gmtime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

#define BST (+1)
#define CCT (+8)

int main ()
{

   time_t rawtime;
   struct tm *info;

   time(&rawtime);
   /* 获取 GMT 时间 */
   info = gmtime(&rawtime );

   printf("当前的世界时钟：\n");
   printf("伦敦：%2d:%02d\n", (info->tm_hour+BST)%24, info->tm_min);
   printf("中国：%2d:%02d\n", (info->tm_hour+CCT)%24, info->tm_min);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前的世界时钟：
伦敦：14:10
中国：21:10
```

C 库函数 - localtime()

描述

C 库函数 **struct tm *localtime(const time_t *timer)** 使用 timer 的值来填充 **tm** 结构。**timer** 的值被分解为 **tm** 结构，并用本地时区表示。

声明

下面是 localtime() 函数的声明。

```
struct tm *localtime(const time_t *timer)
```

参数

- **timer** -- 这是指向表示日历时间的 time_t 值的指针。

返回值

该函数返回指向 **tm** 结构的指针，该结构带有被填充的时间信息。下面是 tm 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

实例

下面的实例演示了 localtime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );
   printf("当前的本地时间和日期：%s", asctime(info));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前的本地时间和日期：Thu Aug 23 09:12:05 2012
```

C 库函数 - mktime()

描述

C 库函数 **time_t mktime(struct tm *timeptr)** 把 **timeptr** 所指向的结构转换为一个依据本地时区的 time_t 值。

声明

下面是 mktime() 函数的声明。

```
time_t mktime(struct tm *timeptr)
```

参数

- **timeptr** -- 这是指向表示日历时间的 time_t 值的指针，该日历时间被分解为以下各部分。下面是 timeptr 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

返回值

该函数返回一个 time_t 值，该值对应于以参数传递的日历时间。如果发生错误，则返回 -1 值。

实例

下面的实例演示了 mktime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   int ret;
   struct tm info;
   char buffer[80];

   info.tm_year = 2001 - 1900;
   info.tm_mon = 7 - 1;
   info.tm_mday = 4;
   info.tm_hour = 0;
   info.tm_min = 0;
   info.tm_sec = 1;
   info.tm_isdst = -1;

   ret = mktime(&info);
   if( ret == -1 )
   {
       printf("错误：不能使用 mktime 转换时间。\n");
   }
   else
   {
      strftime(buffer, sizeof(buffer), "%c", &info );
      print(buffer);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Wed Jul 4 00:00:01 2001
```

C 库函数 - strftime()

描述

C 库函数 **size_t strftime(char *str, size_t maxsize, const char* **format, const struct tm *timeptr)** 根据 **format** 中定义的格式化规则，格式化结构 **timeptr** 表示的时间，并把它存储在 **str** 中。

声明

下面是 strftime() 函数的声明。

```
size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)
```

参数

- **str** -- 这是指向目标数组的指针，用来复制产生的 C 字符串。
- **maxsize** -- 这是被复制到 str 的最大字符数。
- **format** -- 这是 C 字符串，包含了普通字符和特殊格式说明符的任何组合。这些格式说明符由函数替换为表示 tm 中所指定时间的相对应值。格式说明符是：

说明符	替换为	实例
%a	缩写的星期几名称	Sun
%A	完整的星期几名称	Sunday
%b	缩写的月份名称	Mar
%B	完整的月份名称	March
%c	日期和时间表示法	Sun Aug 19 02:56:02 2012
%d	一月中的第几天（01-31）	19
%H	24 小时格式的小时（00-23）	14
%I	12 小时格式的小时（01-12）	05
%j	一年中的第几天（001-366）	231
%m	十进制数表示的月份（01-12）	08
%M	分（00-59）	55
%p	AM 或 PM 名称	PM
%S	秒（00-61）	02
%U	一年中的第几周，以第一个星期日作为第一周的第一天（00-53）	33
%w	十进制数表示的星期几，星期日表示为 0（0-6）	4
%W	一年中的第几周，以第一个星期一作为第一周的第一天（00-53）	34
%x	日期表示法	08/19/12
%X	时间表示法	02:50:06
%y	年份，最后两个数字（00-99）	01
%Y	年份	2012
%Z	时区的名称或缩写	CDT
%%	一个 % 符号	%

- **timeptr** -- 这是指向 tm 结构的指针，该结构包含了一个呗分解为以下各部分的日历时间：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

返回值

如果产生的 C 字符串小于 size 个字符（包括空结束字符），则会返回复制到 str 中的字符总数（不包括空结束字符），否则返回零。

实例

下面的实例演示了 strftime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );

   strftime(buffer,80,"%x - %I:%M%p", info);
   printf("格式化的日期 & 时间 : |%s|\n", buffer );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
格式化的日期 & 时间 : |08/23/12 - 12:40AM|
```

C 库函数 - time()

描述

C 库函数 **time_t time(time_t *seconds)** 返回自纪元 Epoch（1970-01-01 00:00:00 UTC）起经过的时间，以秒为单位。如果 **seconds** 不为空，则返回值也存储在变量 **seconds** 中。

声明

下面是 time() 函数的声明。

```
time_t time(time_t *t)
```

参数

- **seconds** -- 这是指向类型为 time_t 的对象的指针，用来存储 seconds 的值。

返回值

以 time_t 对象返回当前日历时间。

实例

下面的实例演示了 time() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t seconds;

  seconds = time(NULL);
  printf("自 1970-01-01 起的小时数 = %ld\n", seconds/3600);

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
自 1970-01-01 起的小时数 = 373711
```

W3School C++ 教程

来源：C++ 教程

整理：飞龙

C++ 入门

C++ 简介

C++ 是一种静态类型的、编译式的、通用的、大小写敏感的、不规则的编程语言，支持过程化编程、面向对象编程和泛型编程。

C++ 被认为是一种中级语言，它综合了高级语言和低级语言的特点。

C++ 是由 Bjarne Stroustrup 于 1979 年在新泽西州美利山贝尔实验室开始设计开发的。C++ 进一步扩充和完善了 C 语言，最初命名为带类的C，后来在 1983 年更名为 C++。

C++ 是 C 的一个超集，事实上，任何合法的 C 程序都是合法的 C++ 程序。

注意：使用静态类型的编程语言是在编译时执行类型检查，而不是在运行时执行类型检查。

面向对象程序设计

C++ 完全支持面向对象的程序设计，包括面向对象开发的四大特性：

- 封装
- 数据隐藏
- 继承
- 多态

标准库

标准的 C++ 由三个重要部分组成：

- 核心语言，提供了所有构件块，包括变量、数据类型和常量，等等。
- C++ 标准库，提供了大量的函数，用于操作文件、字符串等。
- 标准模板库（STL），提供了大量的方法，用于操作数据结构等。

ANSI 标准

ANSI 标准时为了确保 C++ 的便携性 —— 您所编写的代码在 Mac、UNIX、Windows、Alpha 计算机上都能通过编译。

由于 ANSI 标准已稳定使用了很长的时间，所有主要的 C++ 编译器的制造商都支持 ANSI 标准。

学习 C++

学习 C++，关键是要理解概念，而不应过于深究语言的技术细节。

学习程序设计语言的目的是为了成为一个更好的程序员，也就是说，是为了能更有效率地设计和实现新系统，以及维护旧系统。

C++ 支持多种编程风格。您可以使用 Fortran、C、Smalltalk 等任意一种语言的编程风格来编写代码。每种风格都能有效地保证运行时间效率和空间效率。

C++ 的使用

基本上每个应用程序领域的程序员都有使用 C++。

C++ 通常用于编写设备驱动程序和其他要求实时性的直接操作硬件的软件。

C++ 广泛用于教学和研究。

任何一个使用苹果电脑或 Windows PC 机的用户都在间接地使用 C++，因为这些系统的主要用户接口是使用 C++ 编写的。

C++ 环境设置

本地环境设置

如果您想要设置 C++ 语言环境，您需要确保电脑上有以下两款可用的软件，文本编辑器和 C++ 编译器。

文本编辑器

这将用于输入您的程序。文本编辑器包括 Windows Notepad、OS Edit command、Brief、Epsilon、EMACS 和 vim/vi。

文本编辑器的名称和版本在不同的操作系统上可能会有所不同。例如，Notepad 通常用于 Windows 操作系统上，vim/vi 可用于 Windows 和 Linux/UNIX 操作系统上。

通过编辑器创建的文件通常称为源文件，源文件包含程序源代码。C++ 程序的源文件通常使用扩展名 .cpp、.cp 或 .c。

在开始编程之前，请确保您有一个文本编辑器，且有足够的经验来编写一个计算机程序，然后把它保存在一个文件中，编译并执行它。

C++ 编译器

写在源文件中的源代码是人类可读的源。它需要"编译"，转为机器语言，这样 CPU 可以按给定指令执行程序。

C++ 编译器用于把源代码编译成最终的可执行程序。

大多数的 C++ 编译器并不在乎源文件的扩展名，但是如果您未指定扩展名，则默认使用 .cpp。

最常用的免费可用的编译器是 GNU 的 C/C++ 编译器，如果您使用的是 HP 或 Solaris，则可以使用各自操作系统上的编译器。

以下部分将指导您如何在不同的操作系统上安装 GNU 的 C/C++ 编译器。这里同时提到 C/C++，主要是因为 GNU 的 gcc 编译器适合于 C 和 C++ 编程语言。

安装 GNU 的 C/C++ 编译器

UNIX/Linux 上的安装

如果您使用的是 **Linux** 或 **UNIX**，请在命令行使用下面的命令来检查您的系统上是否安装了 GCC：

```
$ g++ -v
```

如果您的计算机上已经安装了 GNU 编译器，则会显示如下消息：

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

如果未安装 GCC，那么请按照 http://gcc.gnu.org/install/ 上的详细说明安装 GCC。

Mac OS X 上的安装

如果您使用的是 Mac OS X，最快捷的获取 GCC 的方法是从苹果的网站上下载 Xcode 开发环境，并按照安装说明进行安装。一旦安装上 Xcode，您就能使用 GNU 编译器。

Xcode 目前可从 developer.apple.com/technologies/tools/ 上下载。

Windows 上的安装

为了在 Windows 上安装 GCC，您需要安装 MinGW。为了安装 MinGW，请访问 MinGW 的主页 www.mingw.org，进入 MinGW 下载页面，下载最新版本的 MinGW 安装程序，命名格式为 MinGW-.exe。

当安装 MinWG 时，您至少要安装 gcc-core、gcc-g++、binutils 和 MinGW runtime，但是一般情况下都会安装更多其他的项。

添加您安装的 MinGW 的 bin 子目录到您的 **PATH** 环境变量中，这样您就可以在命令行中通过简单的名称来指定这些工具。

当完成安装时，您可以从 Windows 命令行上运行 gcc、g++、ar、ranlib、dlltool 和其他一些 GNU 工具。

C++ 基本语法

C++ 程序可以定义为对象的集合，这些对象通过调用彼此的方法进行交互。现在让我们简要地看一下什么是类、对象，方法、即时变量。

- 对象 **-** 对象具有状态和行为。例如：一只狗的状态 - 颜色、名称、品种，行为 - 摇动、叫唤、吃。对象是类的实例。
- 类 **-** 类可以定义为描述对象行为/状态的模板/蓝图。
- 方法 **-** 从基本上说，一个方法表示一种行为。一个类可以包含多个方法。可以在方法中写入逻辑、操作数据以及执行所有的动作。
- 即时变量 **-** 每个对象都有其独特的即时变量。对象的状态是由这些即时变量的值创建的。

C++ 程序结构

让我们看一段简单的代码，可以输出单词 *Hello World*。

```
#include <iostream>
using namespace std;

// main() 是程序开始执行的地方

int main()
{
   cout << "Hello World"; // 输出 Hello World
   return 0;
}
```

接下来我们讲解一下上面这段程序：

- C++ 语言定义了一些头文件，这些头文件包含了程序中必需的或有用的信息。上面这段程序中，包含了头文件 。
- 行 **using namespace std;** 告诉编译器使用 std 命名空间。命名空间是 C++ 中一个相对新的概念。
- 下一行 **// main()** 是程序开始执行的地方 是一个单行注释。单行注释以 // 开头，在行末结束。
- 下一行 **int main()** 是主函数，程序从这里开始执行。
- 下一行 **cout << "Hello World";** 会在屏幕上显示消息 "Hello World"。
- 下一行 **return 0;** 终止 main( )函数，并向调用进程返回值 0。

编译 & 执行 C++ 程序

接下来让我们看看如何把源代码保存在一个文件中，以及如何编译并运行它。下面是简单的步骤：

- 打开一个文本编辑器，添加上述代码。
- 保存文件为 hello.cpp。
- 打开命令提示符，进入到保存文件所在的目录。
- 键入 'g++ hello.cpp '，输入回车，编译代码。如果代码中没有错误，命令提示符会跳到下一行，并生成 a.out 可执行文件。
- 现在，键入 ' a.out' 来运行程序。
- 您可以看到屏幕上显示 ' Hello World '。

```
$ g++ hello.cpp
$ ./a.out
Hello World
```

请确保您的路径中已包含 g++ 编译器，并确保在包含源文件 hello.cpp 的目录中运行它。

您也可以使用 makefile 来编译 C/C++ 程序。

C++ 中的分号 & 块

在 C++ 中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。

例如，下面是三个不同的语句：

```
x = y;
y = y+1;
add(x, y);
```

块是一组使用大括号括起来的按逻辑连接的语句。例如：

```
{
   cout << "Hello World"; // 输出 Hello World
   return 0;
}
```

C++ 不以行末作为结束符的标识，因此，您可以在一行上放置多个语句。例如：

```
x = y;
y = y+1;
add(x, y);
```

等同于

```
x = y; y = y+1; add(x, y);
```

C++ 标识符

C++ 标识符是用来标识变量、函数、类、模块，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 _ 开始，后跟零个或多个字母、下划线和数字（0-9）。

C++ 标识符内不允许出现标点字符，比如 @、$ 和 %。C++ 是区分大小写的编程语言。因此，在 C++ 中，**Manpower** 和 **manpower** 是两个不同的标识符。

下面列出几个有效的标识符：

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

C++ 关键字

下表列出了 C++ 中的保留字。这些保留字不能作为常量名、变量名或其他标识符名称。

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

三字符组

三字符组就是用于表示另一个字符的三个字符序列，又称为三字符序列。三字符序列总是以两个问号开头。

三字符序列不太常见，但 C++ 标准允许把某些字符指定为三字符序列。以前为了表示键盘上没有的字符，这是必不可少的一种方法。

三字符序列可以出现在任何地方，包括字符串、字符序列、注释和预处理指令。

下面列出了最常用的三字符序列：

三字符组	替换
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

所有的编译器都不支持三字符组，为避免造成混乱，不建议使用三字符组。

C++ 中的空格

只包含空格的行，被称为空白行，可能带有注释，C++ 编译器会完全忽略它。

在 C++ 中，空格用于描述空白符、制表符、换行符和注释。空格分隔语句的各个部分，让编译器能识别语句中的某个元素（比如 int）在哪里结束，下一个元素在哪里开始。因此，在下面的语句中：

```
int age;
```

在这里，int 和 age 之间必须至少有一个空格字符（通常是一个空白符），这样编译器才能够区分它们。另一方面，在下面的语句中：

```
fruit = apples + oranges;   // 获取水果的总数
```

fruit 和 =，或者 = 和 apples 之间的空格字符不是必需的，但是为了增强可读性，您可以根据需要适当增加一些空格。

C++ 注释

程序的注释是解释性语句，您可以在 C++ 代码中包含注释，这将提高源代码的可读性。所有的编程语言都允许某种形式的注释。

C++ 支持单行注释和多行注释。注释中的所有字符会被 C++ 编译器忽略。

C++ 注释以 / 开始，以 / 终止。例如：

```
/* 这是注释 */

/* C++ 注释也可以
 * 跨行
 */
```

注释也能以 // 开始，直到行末为止。例如：

```
#include <iostream>
using namespace std;

main()
{
   cout << "Hello World"; // 输出 Hello World

   return 0;
}
```

当上面的代码被编译时，编译器会忽略 **// prints Hello World**，最后会产生以下结果：

```
Hello World
```

在 / 和 / 注释内部，// 字符没有特殊的含义。在 // 注释内，/ 和 / 字符也没有特殊的含义。因此，您可以在一种注释内嵌套另一种注释。例如：

```
/* 用于输出 Hello World 的注释

cout << "Hello World"; // 输出 Hello World

*/
```

C++ 数据类型

使用编程语言进行编程时，需要用到各种变量来存储各种信息。变量保留的是它所存储的值的内存位置。这意味着，当您创建一个变量时，就会在内存中保留一些空间。

您可能需要存储各种数据类型（比如字符型、宽字符型、整型、浮点型、双浮点型、布尔型等）的信息，操作系统会根据变量的数据类型，来分配内存和决定在保留内存中存储什么。

基本的内置类型

C++ 为程序员提供了种类丰富的内置数据类型和用户自定义的数据类型。下表列出了七种基本的 C++ 数据类型：

类型	关键字
布尔型	bool
字符型	char
整型	int
浮点型	float
双浮点型	double
无类型	void
宽字符型	wchar_t

一些基本类型可以使用一个或多个类型修饰符进行修饰：

- signed
- unsigned
- short
- long

下表显示了各种变量类型在内存中存储值时需要占用的内存，以及该类型的变量所能存储的最大值和最小值。

类型	位宽度	范围
char	1 个字节	-127 到 127 或者 0 到 255
unsigned char	1 个字节	0 到 255
signed char	1 个字节	-127 到 127
int	4 个字节	-2147483648 到 2147483647
unsigned int	4 个字节	0 到 4294967295
signed int	4 个字节	-2147483648 到 2147483647
short int	2 个字节	-32768 到 32767
unsigned short int	Range	0 到 65,535
signed short int	Range	-32768 到 32767
long int	4 个字节	-2,147,483,647 到 2,147,483,647
signed long int	4 个字节	与 long int 相同
unsigned long int	4 个字节	0 到 4,294,967,295
float	4 个字节	+/- 3.4e +/- 38 (~7 个数字)
double	8 个字节	+/- 1.7e +/- 308 (~15 个数字)
long double	8 个字节	+/- 1.7e +/- 308 (~15 个数字)
wchar_t	2 或 4 个字节	1 个宽字符

从上表可得知，变量的大小会根据编译器和所使用的电脑而有所不同。

下面实例会输出您电脑上各种数据类型的大小。

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
   return 0;
}
```

本实例使用了 **endl**，这将在每一行后插入一个换行符，<< 运算符用于向屏幕传多个值。我们也使用 **sizeof()** 函数来获取各种数据类型的大小。

当上面的代码被编译和执行时，它会产生以下的结果，结果会根据所使用的计算机而有所不同：

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

typedef 声明

您可以使用 **typedef** 为一个已有的类型取一个新的名字。下面是使用 typedef 定义一个新类型的语法：

```
typedef type newname;
```

例如，下面的语句会告诉编译器，feet 是 int 的另一个名称：

```
typedef int feet;
```

现在，下面的声明是完全合法的，它创建了一个整型变量 distance：

```
feet distance;
```

枚举类型

枚举类型声明一个可选的类型名称和一组标识符，用来作为该类型的值。的带有零个或多个标识符可以被用来作为该类型的值。每个枚举数是一个枚举类型的常数。

创建枚举，需要使用关键字 **enum**。枚举类型的一般形式为：

```
enum enum-name { list of names } var-list;
```

在这里，enum-name 是枚举类型的名称。名称列表 { list of names } 是用逗号分隔的。

例如，下面的代码定义了一个颜色枚举，变量 c 的类型为 color。最后，c 被赋值为 "blue"。

```
enum color { red, green, blue } c;
c = blue;
```

默认情况下，第一个名称的值为 0，第二个名称的值为 1，第三个名称的值为 2，以此类推。但是，您也可以给名称赋予一个特殊的值，只需要添加一个初始值即可。例如，在下面的枚举中，**green** 的值为 5。

```
enum color { red, green=5, blue };
```

在这里，**blue** 的值为 6，因为默认情况下，每个名称都会比它前面一个名称大 1。

C++ 变量类型

变量其实只不过是程序可操作的存储区的名称。C++ 中每个变量都有指定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可应用于变量上。

变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。大写字母和小写字母是不同的，因为 C++ 是大小写敏感的。

基于前一章讲解的基本类型，有以下几种基本的变量类型，将在下一章中进行讲解：

类型	描述
bool	存储值 true 或 false。
char	通常是一个八位字节（一个字节）。这是一个整数类型。
int	对机器而言，整数的最自然的大小。
float	单精度浮点值。
double	双精度浮点值。
void	表示类型的缺失。
wchar_t	宽字符类型。

C++ 也允许定义各种其他类型的变量，比如枚举、指针、数组、引用、数据结构、类等等，这将会在后续的章节中进行讲解。

下面我们将讲解如何定义、声明和使用各种类型的变量。

C++ 中的变量定义

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

```
type variable_list;
```

在这里，**type** 必须是一个有效的 C++ 数据类型，可以是 char、w_char、int、float、double、bool 或任何用户自定义的对象，**variable_list** 可以由一个或多个标识符名称组成，多个标识符之间用逗号分隔。下面列出几个有效的声明：

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

行 **int i, j, k;** 声明并定义了变量 i、j 和 k，这指示编译器创建类型为 int 的名为 i、j、k 的变量。

变量可以在声明的时候被初始化（指定一个初始值）。初始化器由一个等号，后跟一个常量表达式组成，如下所示：

```
type variable_name = value;
```

下面列举几个实例：

```
extern int d = 3, f = 5;    // d 和 f 的声明
int d = 3, f = 5;           // 定义并初始化 d 和 f
byte z = 22;                // 定义并初始化 z
char x = 'x';               // 变量 x 的值为 'x'
```

不带初始化的定义：带有静态存储持续时间的变量会被隐式初始化为 NULL（所有字节的值都是 0），其他所有变量的初始值是未定义的。

C++ 中的变量声明

变量声明向编译器保证变量以给定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

当您使用多个文件且只在其中一个文件中定义变量时（定义变量的文件在程序连接时是可用的），变量声明就显得非常有用。您可以使用 **extern** 关键字在任何地方声明一个变量。虽然您可以在 C++ 程序中多次声明一个变量，但变量只能在某个文件、函数或代码块中被定义一次。

实例

尝试下面的实例，其中，变量在头部就已经被声明，但它们是在主函数内被定义和初始化的：

```
#include <iostream>
using namespace std;

// 变量声明
extern int a, b;
extern int c;
extern float f;

int main ()
{
  // 变量定义
  int a, b;
  int c;
  float f;

  // 实际初始化
  a = 10;
  b = 20;
  c = a + b;

  cout << c << endl ;

  f = 70.0/3.0;
  cout << f << endl ;

  return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
30
23.3333
```

同样的，在函数声明时，提供一个函数名，而函数的实际定义则可以在任何地方进行。例如：

```
// 函数声明
int func();

int main()
{
    // 函数调用
    int i = func();
}

// 函数定义
int func()
{
    return 0;
}
```

C++ 中的左值（Lvalues）和右值（Rvalues）

C++ 中有两种类型的表达式：

- 左值（**lvalue**）：指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边。
- 右值（**rvalue**）：术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能

对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边。下面是一个有效的语句：

```
int g = 20;
```

但是下面这个就不是一个有效的语句，会生成编译时错误：

```
10 = 20;
```

C++ 变量作用域

作用域是程序的一个区域，一般来说有三个地方可以声明变量：

- 在函数或一个代码块内部声明的变量，称为局部变量。
- 在函数参数的定义中声明的变量，称为形式参数。
- 在所有函数外部声明的变量，称为全局变量。

我们将在后续的章节中学习什么是函数和参数。本章我们先来讲解声明是局部变量和全局变量。

局部变量

在函数或一个代码块内部声明的变量，称为局部变量。它们只能被函数内部或者代码块内部的语句使用。下面的实例使用了局部变量：

```
#include <iostream>
using namespace std;

int main ()
{
  // 局部变量声明
  int a, b;
  int c;

  // 实际初始化
  a = 10;
  b = 20;
  c = a + b;

  cout << c;

  return 0;
}
```

全局变量

在所有函数外部定义的变量（通常是在程序的头部），称为全局变量。全局变量的值在程序的整个生命周期内都是有效的。

全局变量可以被任何函数访问。也就是说，全局变量一旦声明，在整个程序中都是可用的。下面的实例使用了全局变量和局部变量：

```
#include <iostream>
using namespace std;

// 全局变量声明
int g;

int main ()
{
  // 局部变量声明
  int a, b;

  // 实际初始化
  a = 10;
  b = 20;
  g = a + b;

  cout << g;

  return 0;
}
```

在程序中，局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值。下面是一个实例：

```
#include <iostream>
using namespace std;

// 全局变量声明
int g = 20;

int main ()
{
  // 局部变量声明
  int g = 10;

  cout << g;

  return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
10
```

初始化局部变量和全局变量

当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动初始化为下列值：

数据类型	初始化默认值
int	0
char	'\0'
float	0
double	0
pointer	NULL

正确地初始化变量是一个良好的编程习惯，否则有时候程序可能会产生意想不到的结果。

C++ 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量。

常量可以是任何的基本数据类型，可分为整型数字、浮点数字、字符、字符串和布尔值。

常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

```
212         // 合法的
215u        // 合法的
0xFeeL      // 合法的
078         // 非法的：8 不是八进制的数字
032UU       // 非法的：不能重复后缀
```

以下是各种类型的整数常量的实例：

```
85         // 十进制
0213       // 八进制
0x4b       // 十六进制
30         // 整数
30u        // 无符号整数
30l        // 长整数
30ul       // 无符号长整数
```

浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含小数点、指数，或同时包含两者。当使用指数形式表示时，必须包含整数部分、小数部分，或同时包含两者。带符号的指数是用 e 或 E 引入的。

下面列举几个浮点常量的实例：

```
3.14159       // 合法的
314159E-5L    // 合法的
510E          // 非法的：不完整的指数
210f          // 非法的：没有小数或指数
.e55          // 非法的：缺少整数或分数
```

布尔常量

布尔常量共有两个，它们都是标准的 C++ 关键字：

- **true** 值代表真。
- **false** 值代表假。

我们不应把 true 的值看成 1，把 false 的值看成 0。

字符常量

字符常量是括在单引号中。如果常量以 L（仅当大写时）开头，则表示它是一个宽字符常量（例如 L'x'），此时它必须存储在 **wchar_t** 类型的变量中。否则，它就是一个窄字符常量（例如 'x'），此时它可以存储在 **char** 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）。

在 C++ 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：

转义序列	含义
\	\ 字符
\'	' 字符
\"	" 字符
\?	? 字符
\a	警报铃声
\b	退格键
\f	换页符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ooo	一到三位的八进制数
\xhh . . .	一个或多个数字的十六进制数

下面的实例显示了一些转义序列字符：

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello\tWorld\n\n";
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello   World
```

字符串常量

字符串字面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

定义常量

在 C++ 中，有两种简单的定义常量的方式：

- 使用 **#define** 预处理器。
- 使用 **const** 关键字。

#define 预处理器

下面是使用 #define 预处理器定义常量的形式：

```
#define identifier value
```

具体请看下面的实例：

```
#include <iostream>
using namespace std;

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main()
{

   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
50
```

const 关键字

您可以使用 **const** 前缀声明指定类型的常量，如下所示：

```
const type variable = value;
```

具体请看下面的实例：

```
#include <iostream>
using namespace std;

int main()
{
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
50
```

请注意，把常量定义为大写字母形式，是一个很好的编程实践。

C++ 修饰符类型

C++ 允许在 **char**、**int** 和 **double** 数据类型前放置修饰符。修饰符用于改变基本类型的含义，所以它更能满足各种情境的需求。

下面列出了数据类型修饰符：

- signed
- unsigned
- long
- short

修饰符 **signed**、**unsigned**、**long** 和 **short** 可应用于整型，**signed** 和 **unsigned** 可应用于字符型，**long** 可应用于双精度型。

修饰符 **signed** 和 **unsigned** 也可以作为 **long** 或 **short** 修饰符的前缀。例如：**unsigned long int**。

C++ 允许使用速记符号来声明无符号短整数或无符号长整数。您可以不写 int，只写单词 **unsigned**、**short** 或 **unsigned**、**long**，int 是隐含的。例如，下面的两个语句都声明了无符号整型变量。

```
unsigned x;
unsigned int y;
```

为了理解 C++ 解释有符号整数和无符号整数修饰符之间的差别，我们来运行一下下面这个短程序：

```
#include <iostream>
using namespace std;

/*
 * 这个程序演示了有符号整数和无符号整数之间的差别
*/
int main()
{
   short int i;           // 有符号短整数
   short unsigned int j;  // 无符号短整数

   j = 50000;

   i = j;
   cout << i << " " << j;

   return 0;
}
```

当上面的程序运行时，会输出下列结果：

```
-15536 50000
```

上述结果中，无符号短整数 50,000 的位模式被解释为有符号短整数 -15,536。

C++ 中的类型限定符

类型限定符提供了变量的额外信息。

限定符	含义
const	**const** 类型的对象在程序执行期间不能被修改改变。
volatile	修饰符 **volatile** 告诉编译器，变量的值可能以程序未明确指定的方式被改变。
restrict	由 **restrict** 修饰的指针是唯一一种访问它所指向的对象的方式。只有 C99 增加了新的类型限定符 restrict。

C++ 存储类

存储类定义 C++ 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C++ 程序中可用的存储类：

- auto
- register
- static
- extern
- mutable

auto 存储类

auto 存储类是所有局部变量默认的存储类。

```
{
   int mount;
   auto int month;
}
```

上面的实例定义了两个带有相同存储类的变量，auto 只能用在函数内，即 auto 只能修饰局部变量。

register 存储类

register 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

```
{
   register int  miles;
}
```

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

static 存储类

static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。

static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

在 C++ 中，当 static 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

```
#include <iostream>

// 函数声明
void func(void);

static int count = 10; /* 全局变量 */

main()
{
    while(count--)
    {
       func();
    }
    return 0;
}
// 函数定义
void func( void )
{
    static int i = 5; // 局部静态变量
    i++;
    std::cout << "i is " << i ;
    std::cout << " and count is " << count << std::endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

extern 存储类

extern 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 *extern* 来得到已定义的变量或函数的引用。可以这么理解，*extern* 是用来在另一个文件中声明一个全局变量或函数。

extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

第一个文件：main.cpp

```
#include <iostream>

int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

第二个文件：support.cpp

```
#include <iostream>

extern int count;

void write_extern(void)
{
   std::cout << "Count is " << count << std::endl;
}
```

在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.cpp 中定义的 count。现在 ，编译这两个文件，如下所示：

```
$g++ main.cpp support.cpp -o write
```

这会产生 **write** 可执行程序，尝试执行 **write**，它会产生下列结果：

```
$./write
5
```

mutable 存储类

mutable 说明符仅适用于类的对象，这将在本教程的最后进行讲解。它允许对象的成员替代常量。也就是说，mutable 成员可以通过 const 成员函数修改。

C++ 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C++ 内置了丰富的运算符，并提供了以下类型的运算符：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

本章将逐一介绍算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符和其他运算符。

算术运算符

下表显示了 C++ 支持的所有算术运算符。

假设变量 A 的值为 10，变量 B 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

实例

请看下面的实例，了解 C++ 中所有可用的算术运算符。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

```
#include <iostream>
using namespace std;

main()
{
   int a = 21;
   int b = 10;
   int c ;

   c = a + b;
   cout << "Line 1 - c 的值是 " << c << endl ;
   c = a - b;
   cout << "Line 2 - c 的值是 " << c << endl ;
   c = a * b;
   cout << "Line 3 - c 的值是 " << c << endl ;
   c = a / b;
   cout << "Line 4 - c 的值是 " << c << endl ;
   c = a % b;
   cout << "Line 5 - c 的值是 " << c << endl ;
   c = a++;
   cout << "Line 6 - c 的值是 " << c << endl ;
   c = a--;
   cout << "Line 7 - c 的值是 " << c << endl ;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 21
Line 7 - c 的值是 22
```

关系运算符

下表显示了 C++ 支持的所有关系运算符。

假设变量 A 的值为 10，变量 B 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

实例

请看下面的实例，了解 C++ 中所有可用的关系运算符。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

```
#include <iostream>
using namespace std;

main()
{
   int a = 21;
   int b = 10;
   int c ;

   if( a == b )
   {
      cout << "Line 1 - a 等于 b" << endl ;
   }
   else
   {
      cout << "Line 1 - a 不等于 b" << endl ;
   }
   if ( a < b )
   {
      cout << "Line 2 - a 小于 b" << endl ;
   }
   else
   {
      cout << "Line 2 - a 不小于 b" << endl ;
   }
   if ( a > b )
   {
      cout << "Line 3 - a 大于 b" << endl ;
   }
   else
   {
      cout << "Line 3 - a 不大于 b" << endl ;
   }
   /* 改变 a 和 b 的值 */
   a = 5;
   b = 20;
   if ( a <= b )
   {
      cout << "Line 4 - a 小于或等于 b" << endl ;
   }
   if ( b >= a )
   {
      cout << "Line 5 - b 大于或等于 b" << endl ;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - a 不等于 b
Line 2 - a 不小于 b
Line 3 - a 大于 b
Line 4 - a 小于或等于 b
Line 5 - b 大于或等于 b
```

逻辑运算符

下表显示了 C++ 支持的所有关系逻辑运算符。

假设变量 A 的值为 1，变量 B 的值为 0，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A || B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

实例

请看下面的实例，了解 C++ 中所有可用的逻辑运算符。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

```
#include <iostream>
using namespace std;

main()
{
   int a = 5;
   int b = 20;
   int c ;

   if ( a && b )
   {
      cout << "Line 1 - 条件为真"<< endl ;
   }
   if ( a || b )
   {
      cout << "Line 2 - 条件为真"<< endl ;
   }
   /* 改变 a 和 b 的值 */
   a = 0;
   b = 10;
   if ( a && b )
   {
      cout << "Line 3 - 条件为真"<< endl ;
   }
   else
   {
      cout << "Line 4 - 条件不为真"<< endl ;
   }
   if ( !(a && b) )
   {
      cout << "Line 5 - 条件为真"<< endl ;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

位运算符

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

下表显示了 C++ 支持的位运算符。假设变量 A 的值为 60，变量 B 的值为 13，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果。	(~A) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

请看下面的实例，了解 C++ 中所有可用的位运算符。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

```
#include <iostream>
using namespace std;

main()
{
   unsigned int a = 60;    // 60 = 0011 1100
   unsigned int b = 13;    // 13 = 0000 1101
   int c = 0;

   c = a & b;              // 12 = 0000 1100
   cout << "Line 1 - c 的值是 " << c << endl ;

   c = a | b;              // 61 = 0011 1101
   cout << "Line 2 - c 的值是 " << c << endl ;

   c = a ^ b;              // 49 = 0011 0001
   cout << "Line 3 - c 的值是 " << c << endl ;

   c = ~a;                 // -61 = 1100 0011
   cout << "Line 4 - c 的值是 " << c << endl ;

   c = a << 2;             // 240 = 1111 0000
   cout << "Line 5 - c 的值是 " << c << endl ;

   c = a >> 2;             // 15 = 0000 1111
   cout << "Line 6 - c 的值是 " << c << endl ;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

赋值运算符

下表列出了 C++ 支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C = *A* 相当于 *C* = *C* A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
|=	按位或且赋值运算符	C |= 2 等同于 C = C | 2

实例

请看下面的实例，了解 C++ 中所有可用的赋值运算符。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

```
#include <iostream>
using namespace std;

main()
{
   int a = 21;
   int c ;

   c =  a;
   cout << "Line 1 - =  运算符实例，c 的值 = : " <<c<< endl ;

   c +=  a;
   cout << "Line 2 - += 运算符实例，c 的值 = : " <<c<< endl ;

   c -=  a;
   cout << "Line 3 - -= 运算符实例，c 的值 = : " <<c<< endl ;

   c *=  a;
   cout << "Line 4 - *= 运算符实例，c 的值 = : " <<c<< endl ;

   c /=  a;
   cout << "Line 5 - /= 运算符实例，c 的值 = : " <<c<< endl ;

   c  = 200;
   c %=  a;
   cout << "Line 6 - %= 运算符实例，c 的值 = : " <<c<< endl ;

   c <<=  2;
   cout << "Line 7 - <<= 运算符实例，c 的值 = : " <<c<< endl ;

   c >>=  2;
   cout << "Line 8 - >>= 运算符实例，c 的值 = : " <<c<< endl ;

   c &=  2;
   cout << "Line 9 - &= 运算符实例，c 的值 = : " <<c<< endl ;

   c ^=  2;
   cout << "Line 10 - ^= 运算符实例，c 的值 = : " <<c<< endl ;

   c |=  2;
   cout << "Line 11 - |= 运算符实例，c 的值 = : " <<c<< endl ;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - =  运算符实例，c 的值 = 21
Line 2 - += 运算符实例，c 的值 = 42
Line 3 - -= 运算符实例，c 的值 = 21
Line 4 - *= 运算符实例，c 的值 = 441
Line 5 - /= 运算符实例，c 的值 = 21
Line 6 - %= 运算符实例，c 的值 = 11
Line 7 - <<= 运算符实例，c 的值 = 44
Line 8 - >>= 运算符实例，c 的值 = 11
Line 9 - &= 运算符实例，c 的值 = 2
Line 10 - ^= 运算符实例，c 的值 = 0
Line 11 - |= 运算符实例，c 的值 = 2
```

杂项运算符

下表列出了 C++ 支持的其他一些重要的运算符。

运算符	描述
sizeof	sizeof 运算符返回变量的大小。例如，sizeof(a) 将返回 4，其中 a 是整数。
Condition ? X : Y	条件运算符。如果 Condition 为真 ? 则值为 X : 否则值为 Y。
,	逗号运算符会顺序执行一系列运算。整个逗号表达式的值是以逗号分隔的列表中的最后一个表达式的值。
.（点）和 ->（箭头）	成员运算符用于引用类、结构和共用体的成员。
Cast	强制转换运算符把一种数据类型转换为另一种数据类型。例如，int(2.2000) 将返回 2。
&	指针运算符 & 返回变量的地址。例如 &a; 将给出变量的实际地址。
*	指针运算符 * 指向一个变量。例如，*var; 将指向变量 var。

C++ 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 *2*，在这里，*x* 被赋值为 *13*，而不是 *20*，因为运算符 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	() [] -> . ++ - -	从左到右
一元	+ - ! ~ ++ - - (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

实例

请看下面的实例，了解 C++ 中运算符的优先级。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

对比有括号和没有括号时的区别，这将产生不同的结果。因为 ()、 /、 * 和 + 有不同的优先级，高优先级的操作符将优先计算。

```
#include <iostream>
using namespace std;

main()
{
   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;

   e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   cout << "(a + b) * c / d 的值是 " << e << endl ;

   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   cout << "((a + b) * c) / d 的值是 " << e << endl ;

   e = (a + b) * (c / d);   // (30) * (15/5)
   cout << "(a + b) * (c / d) 的值是 " << e << endl ;

   e = a + (b * c) / d;     //  20 + (150/5)
   cout << "a + (b * c) / d 的值是 " << e << endl ;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

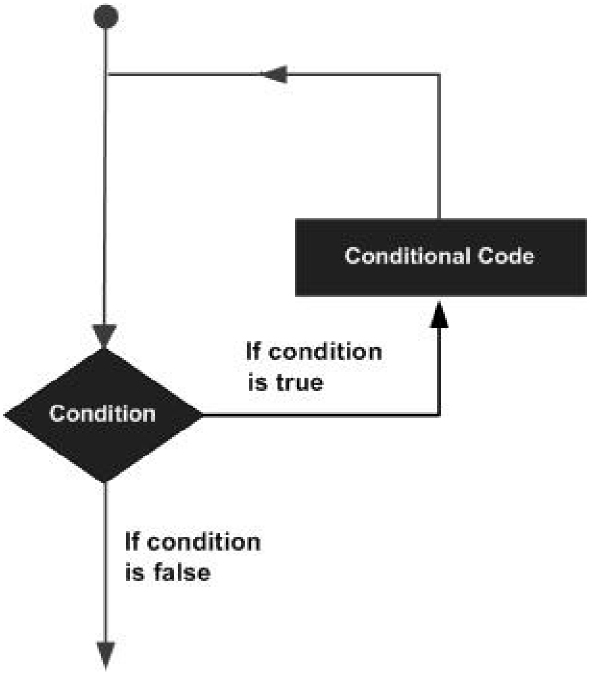
```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

C++ 循环

有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：



循环类型

C++ 编程语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 while、for 或 do..while 循环内使用一个或多个循环。

循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C++ 提供了下列的控制语句。点击链接查看每个语句的细节。

控制语句	描述
break 语句	终止 **loop** 或 **switch** 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。
continue 语句	引起循环跳过主体的剩余部分，立即重新开始测试条件。
goto 语句	将控制转移到被标记的语句。但是不建议在程序中使用 goto 语句。

无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

```
#include <iostream>
using namespace std;

int main ()
{

   for( ; ; )
   {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，C++ 程序员偏向于使用 for(;;) 结构来表示一个无限循环。

注意：您可以按 Ctrl + C 键终止一个无限循环。

C++ while 循环

只要给定的条件为真，**while** 循环语句会重复执行一个目标语句。

语法

C++ 中 **while** 循环的语法：

```
while(condition)
{
   statement(s);
}
```

在这里，**statement(s)** 可以是一个单独的语句，也可以是几个语句组成的代码块。**condition** 可以是任意的表达式，当为任意非零值时都为真。当条件为真时执行循环。

当条件为假时，程序流将继续执行紧接着循环的下一条语句。

流程图

在这里，*while* 循环的关键点是循环可能一次都不会执行。当条件被测试且结果为假时，会跳过循环主体，直接执行紧接着 while 循环的下一条语句。

实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 10;

   // while 循环执行
   while( a < 20 )
   {
       cout << "a 的值：" << a << endl;
       a++;
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

C++ for 循环

for 循环允许您编写一个执行特定次数的循环的重复控制结构。

语法

C++ 中 **for** 循环的语法：

```
for ( init; condition; increment )
{
   statement(s);
}
```

下面是 for 循环的控制流：

1. **init** 会首先被执行，且只会执行一次。这一步允许您声明并初始化任何循环控制变量。您也可以不在这里写任何语句，只要有一个分号出现即可。
2. 接下来，会判断 **condition**。如果为真，则执行循环主体。如果为假，则不执行循环主体，且控制流会跳转到紧接着 for 循环的下一条语句。
3. 在执行完 for 循环主体后，控制流会跳回上面的 **increment** 语句。该语句允许您更新循环控制变量。该语句可以留空，只要在条件后有一个分号出现即可。
4. 条件再次被判断。如果为真，则执行循环，这个过程会不断重复（循环主体，然后增加步值，再然后重新判断条件）。在条件变为假时，for 循环终止。

流程图

实例

```
#include <iostream>
using namespace std;

int main ()
{
   // for 循环执行
   for( int a = 10; a < 20; a = a + 1 )
   {
       cout << "a 的值：" << a << endl;
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

C++ do...while 循环

不像 **for** 和 **while** 循环，它们是在循环头部测试循环条件。**do...while** 循环是在循环的尾部检查它的条件。

do...while 循环与 while 循环类似，但是 do...while 循环会确保至少执行一次循环。

语法

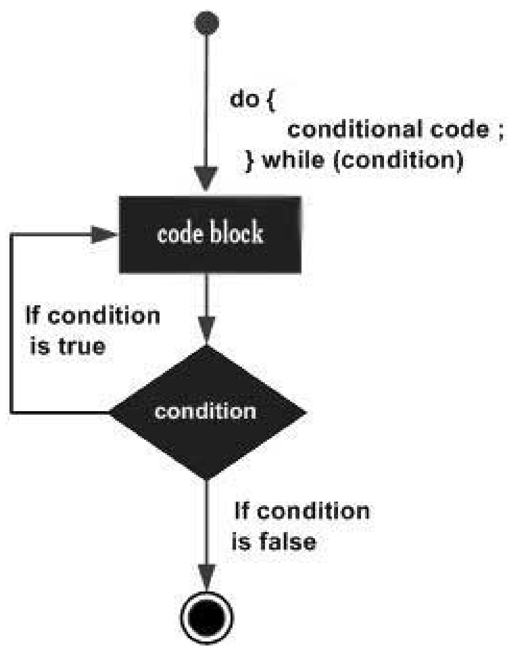
C++ 中 **do...while** 循环的语法：

```
do
{
   statement(s);

}while( condition );
```

请注意，条件表达式出现在循环的尾部，所以循环中的 statement(s) 会在条件被测试之前至少执行一次。

如果条件为真，控制流会跳转回上面的 do，然后重新执行循环中的 statement(s)。这个过程会不断重复，直到给定条件变为假为止。

流程图



实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 10;

   // do 循环执行
   do
   {
       cout << "a 的值：" << a << endl;
       a = a + 1;
   }while( a < 20 );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

C++ 嵌套循环

一个循环内可以嵌套另一个循环。C++ 允许至少 256 个嵌套层次。

语法

C++ 中 嵌套 **for** 循环 语句的语法：

```
for ( init; condition; increment )
{
   for ( init; condition; increment )
   {
      statement(s);
   }
   statement(s); // 可以放置更多的语句
}
```

C++ 中 嵌套 **while** 循环 语句的语法：

```
while(condition)
{
   while(condition)
   {
      statement(s);
   }
   statement(s); // 可以放置更多的语句
}
```

C++ 中 嵌套 **do...while** 循环 语句的语法：

```
do
{
   statement(s); // 可以放置更多的语句
   do
   {
      statement(s);
   }while( condition );

}while( condition );
```

关于嵌套循环有一点值得注意，您可以在任何类型的循环内嵌套其他任何类型的循环。比如，一个 for 循环可以嵌套在一个 while 循环内，反之亦然。

实例

下面的程序使用了一个嵌套的 for 循环来查找 2 到 100 中的质数：

```
#include <iostream>
using namespace std;

int main ()
{
   int i, j;

   for(i=2; i<100; i++) {
      for(j=2; j <= (i/j); j++)
        if(!(i%j)) break; // 如果找到，则不是质数
        if(j > (i/j)) cout << i << " 是质数\n";
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
2 是质数
3 是质数
5 是质数
7 是质数
11 是质数
13 是质数
17 是质数
19 是质数
23 是质数
29 是质数
31 是质数
37 是质数
41 是质数
43 是质数
47 是质数
53 是质数
59 是质数
61 是质数
67 是质数
71 是质数
73 是质数
79 是质数
83 是质数
89 是质数
97 是质数
```

C++ break 语句

C++ 中 **break** 语句有以下两种用法：

1. 当 **break** 语句出现在一个循环内时，循环会立即终止，且程序流将继续执行紧接着循环的下一条语句。
2. 它可用于终止 **switch** 语句中的一个 case。

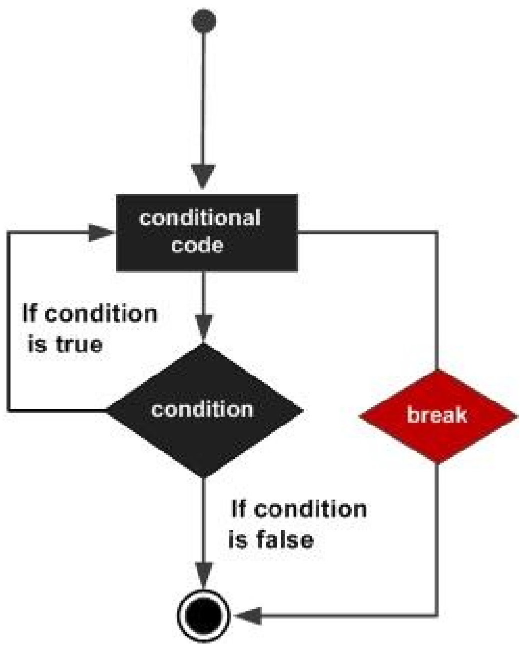
如果您使用的是嵌套循环（即一个循环内嵌套另一个循环），break 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。

语法

C++ 中 **break** 语句的语法：

```
break;
```

流程图



实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 10;

   // do 循环执行
   do
   {
       cout << "a 的值：" << a << endl;
       a = a + 1;
       if( a > 15)
       {
          // 终止循环
          break;
       }
   }while( a < 20 );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
```

C++ continue 语句

C++ 中的 **continue** 语句有点像 **break** 语句。但它不是强迫终止，continue 会跳过当前循环中的代码，强迫开始下一次循环。

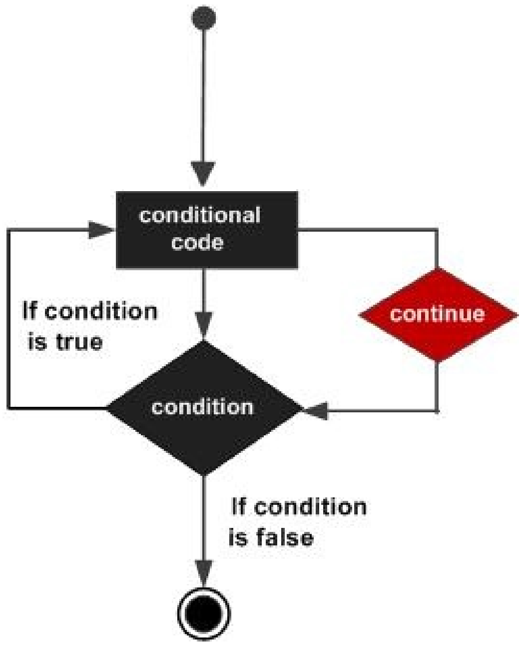
对于 **for** 循环，**continue** 语句会导致执行条件测试和循环增量部分。对于 **while** 和 **do...while** 循环，**continue** 语句会导致程序控制回到条件测试上。

语法

C++ 中 **continue** 语句的语法：

```
continue;
```

流程图



实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 10;

   // do 循环执行
   do
   {
       if( a == 15)
       {
          // 跳过迭代
          a = a + 1;
          continue;
       }
       cout << "a 的值：" << a << endl;
       a = a + 1;
   }while( a < 20 );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

C++ goto 语句

goto 语句允许把控制无条件转移到同一函数内的被标记的语句。

注意：在任何编程语言中，都不建议使用 goto 语句。因为它使得程序的控制流难以跟踪，使程序难以理解和难以修改。任何使用 goto 语句的程序可以改写成不需要使用 goto 语句的写法。

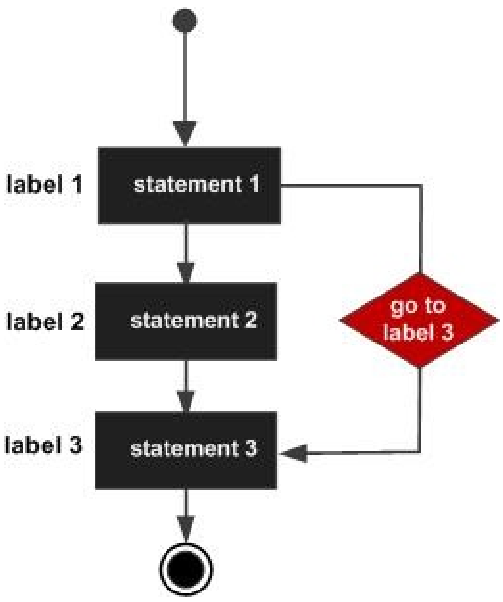
语法

C++ 中 **goto** 语句的语法：

```
goto label;
..
.
label: statement;
```

在这里，**label** 是识别被标记语句的标识符，可以是任何除 C++ 关键字以外的纯文本。标记语句可以是任何语句，放置在标识符和冒号（:）后边。

流程图



实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 10;

   // do 循环执行
   LOOP:do
   {
       if( a == 15)
       {
          // 跳过迭代
          a = a + 1;
          goto LOOP;
       }
       cout << "a 的值：" << a << endl;
       a = a + 1;
   }while( a < 20 );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

goto 语句一个很好的作用是退出深嵌套例程。例如，请看下面的代码片段：

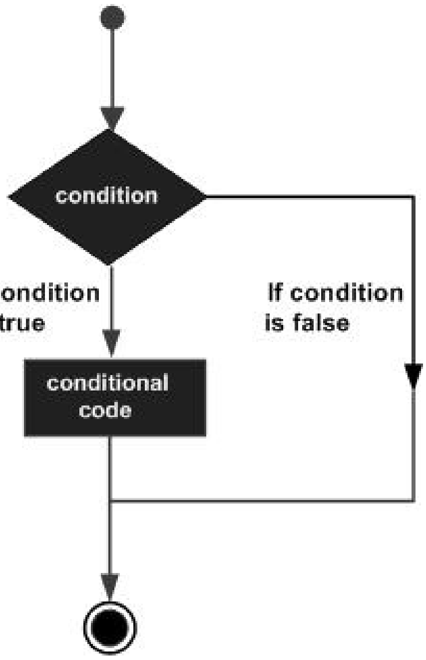
```
for(...) {
   for(...) {
      while(...) {
         if(...) goto stop;
         .
         .
         .
      }
   }
}
stop:
cout << "Error in program.\n";
```

消除 **goto** 会导致一些额外的测试被执行。一个简单的 **break** 语句在这里不会起到作用，因为它只会使程序退出最内层循环。

C++ 判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

下面是大多数编程语言中典型的判断结构的一般形式：



判断语句

C++ 编程语言提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
if 语句	一个 **if 语句** 由一个布尔表达式后跟一个或多个语句组成。
if...else 语句	一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。
嵌套 if 语句	您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。
switch 语句	一个 **switch** 语句允许测试一个变量等于多个值时的情况。
嵌套 switch 语句	您可以在一个 **switch** 语句内使用另一个 **switch** 语句。

? : 运算符

我们已经在前面的章节中讲解了 条件运算符 ? :，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

C++ if 语句

一个 **if** 语句 由一个布尔表达式后跟一个或多个语句组成。

语法

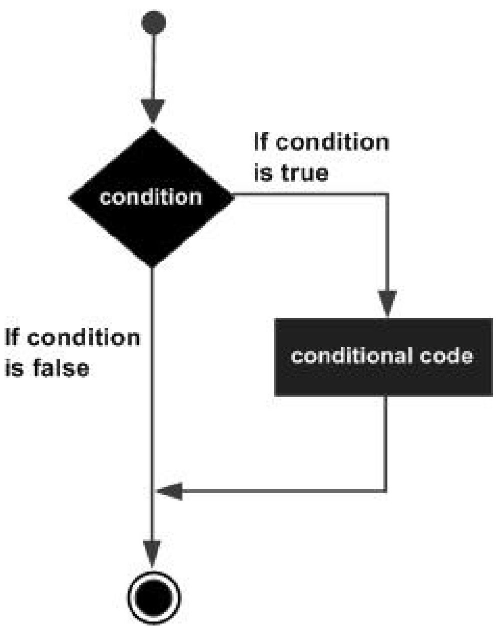
C++ 中 **if** 语句的语法：

```
if(boolean_expression)
{
   // 如果布尔表达式为真将执行的语句
}
```

如果布尔表达式为 **true**，则 if 语句内的代码块将被执行。如果布尔表达式为 **false**，则 if 语句结束后的第一组代码（闭括号后）将被执行。

C 语言把任何非零和非空的值假定为 **true**，把零或 **null** 假定为 **false**。

流程图



实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 10;

   // 使用 if 语句检查布尔条件
   if( a < 20 )
   {
       // 如果条件为真，则输出下面的语句
       cout << "a 小于 20" << endl;
   }
   cout << "a 的值是 " << a << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 小于 20
a 的值是 10
```

C++ if...else 语句

一个 **if** 语句 后可跟一个可选的 **else** 语句，else 语句在布尔表达式为假时执行。

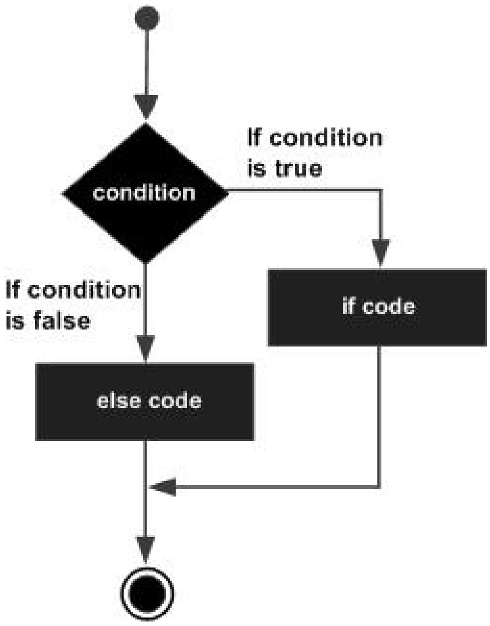
语法

C++ 中 **if...else** 语句的语法：

```
if(boolean_expression)
{
   // 如果布尔表达式为真将执行的语句
}
else
{
   // 如果布尔表达式为假将执行的语句
}
```

如果布尔表达式为 **true**，则执行 **if** 块内的代码。如果布尔表达式为 **false**，则执行 **else** 块内的代码。

流程图



实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 100;

   // 检查布尔条件
   if( a < 20 )
   {
       // 如果条件为真，则输出下面的语句
       cout << "a 小于 20" << endl;
   }
   else
   {
       // 如果条件为假，则输出下面的语句
       cout << "a 大于 20" << endl;
   }
   cout << "a 的值是 " << a << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 大于 20
a 的值是 100
```

if...else if...else 语句

一个 **if** 语句后可跟一个可选的 **else if...else** 语句，这可用于测试多种条件。

当使用 if...else if...else 语句时，以下几点需要注意：

- 一个 if 后可跟零个或一个 else，else 必须在所有 else if 之后。
- 一个 if 后可跟零个或多个 else if，else if 必须在 else 之前。
- 一旦某个 else if 匹配成功，其他的 else if 或 else 将不会被测试。

语法

C++ 中的 **if...else if...else** 语句的语法：

```
if(boolean_expression 1)
{
   // 当布尔表达式 1 为真时执行
}
else if( boolean_expression 2)
{
   // 当布尔表达式 2 为真时执行
}
else if( boolean_expression 3)
{
   // 当布尔表达式 3 为真时执行
}
else
{
   // 当上面条件都不为真时执行
}
```

实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 100;

   // 检查布尔条件
   if( a == 10 )
   {
       // 如果 if 条件为真，则输出下面的语句
       cout << "a 的值是 10" << endl;
   }
   else if( a == 20 )
   {
       // 如果 else if 条件为真，则输出下面的语句
       cout << "a 的值是 20" << endl;
   }
   else if( a == 30 )
   {
       // 如果 else if 条件为真，则输出下面的语句
       cout << "a 的值是 30" << endl;
   }
   else
   {
       // 如果上面条件都不为真，则输出下面的语句
       cout << "没有匹配的值" << endl;
   }
   cout << "a 的准确值是 " << a << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
没有匹配的值
a 的准确值是 100
```

C++ 嵌套 if 语句

在 C++ 中，嵌套 if-else 语句是合法的，这意味着您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。

语法

C++ 中 嵌套 **if** 语句的语法：

```
if( boolean_expression 1)
{
   // 当布尔表达式 1 为真时执行
   if(boolean_expression 2)
   {
      // 当布尔表达式 2 为真时执行
   }
}
```

您可以嵌套 **else if...else**，方式与嵌套 *if* 语句相似。

实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 100;
   int b = 200;

   // 检查布尔条件
   if( a == 100 )
   {
       // 如果条件为真，则检查下面的条件
       if( b == 200 )
       {
          // 如果条件为真，则输出下面的语句
          cout << "a 的值是 100，且 b 的值是 200" << endl;
       }
   }
   cout << "a 的准确值是 " << a << endl;
   cout << "b 的准确值是 " << b << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值是 100，且 b 的值是 200
a 的准确值是 100
b 的准确值是 200
```

C++ switch 语句

一个 **switch** 语句允许测试一个变量等于多个值时的情况。每个值称为一个 case，且被测试的变量会对每个 **switch case** 进行检查。

语法

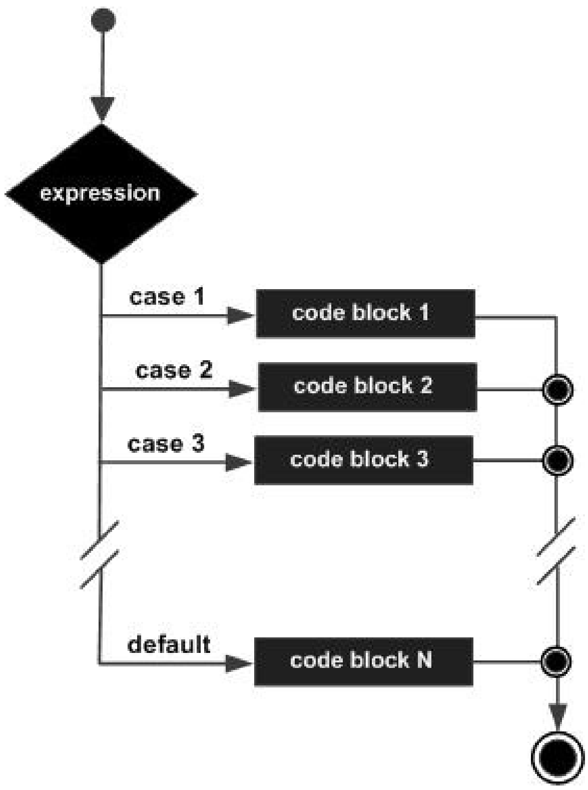
C++ 中 **switch** 语句的语法：

```
switch(expression){
    case constant-expression  :
       statement(s);
       break; // 可选的
    case constant-expression  :
       statement(s);
       break; // 可选的

    // 您可以有任意数量的 case 语句
    default : // 可选的
       statement(s);
}
```

switch 语句必须遵循下面的规则：

- **switch** 语句中的 **expression** 必须是一个整型或枚举类型，或者是一个 class 类型，其中 class 有一个单一的转换函数将其转换为整型或枚举类型。
- 在一个 switch 中可以有任意数量的 case 语句。每个 case 后跟一个要比较的值和一个冒号。
- case 的 **constant-expression** 必须与 switch 中的变量具有相同的数据类型，且必须是一个常量或字面量。
- 当被测试的变量等于 case 中的常量时，case 后跟的语句将被执行，直到遇到 **break** 语句为止。
- 当遇到 **break** 语句时，switch 终止，控制流将跳转到 switch 语句后的下一行。
- 不是每一个 case 都需要包含 **break**。如果 case 语句不包含 **break**，控制流将会 继续 后续的 case，直到遇到 break 为止。
- 一个 **switch** 语句可以有一个可选的 **default** case，出现在 switch 的结尾。default case 可用于在上面所有 case 都不为真时执行一个任务。default case 中的 **break** 语句不是必需的。

流程图



实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   char grade = 'D';

   switch(grade)
   {
   case 'A' :
      cout << "很棒！" << endl;
      break;
   case 'B' :
   case 'C' :
      cout << "做得好" << endl;
      break;
   case 'D' :
      cout << "您通过了" << endl;
      break;
   case 'F' :
      cout << "最好再试一下" << endl;
      break;
   default :
      cout << "无效的成绩" << endl;
   }
   cout << "您的成绩是 " << grade << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
您通过了
您的成绩是 D
```

C++ 嵌套 switch 语句

您可以把一个 **switch** 作为一个外部 **switch** 的语句序列的一部分，即可以在一个 **switch** 语句内使用另一个 **switch** 语句。即使内部和外部 switch 的 case 常量包含共同的值，也没有矛盾。

C++ 中的 switch 语句允许至少 256 个嵌套层次。

语法

C++ 中 嵌套 **switch** 语句的语法：

```
switch(ch1) {
   case 'A':
      cout << "这个 A 是外部 switch 的一部分";
      switch(ch2) {
         case 'A':
            cout << "这个 A 是内部 switch 的一部分";
            break;
         case 'B': // 内部 B case 代码
      }
      break;
   case 'B': // 外部 B case 代码
}
```

实例

```
#include <iostream>
using namespace std;

int main ()
{
   // 局部变量声明
   int a = 100;
   int b = 200;

   switch(a) {
      case 100:
         cout << "这是外部 switch 的一部分" << endl;
         switch(b) {
            case 200:
               cout << "这是内部 switch 的一部分" << endl;
         }
   }
   cout << "a 的准确值是 " << a << endl;
   cout << "b 的准确值是 " << b << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
这是外部 switch 的一部分
这是内部 switch 的一部分
a 的准确值是 100
b 的准确值是 200
```

C++ 函数

函数是一组一起执行一个任务的语句。每个 C++ 程序都至少有一个函数，即主函数 **main()** ，所有简单的程序都可以定义其他额外的函数。

您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

C++ 标准库提供了大量的程序可以调用的内置函数。例如，函数 **strcat()** 用来连接两个字符串，函数 **memcpy()** 用来复制内存到另一个位置。

函数还有很多叫法，比如方法、子例程或程序，等等。

定义函数

C++ 中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
   body of the function
}
```

在 C++ 中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

- 返回类型：一个函数可以返回一个值。**return_type** 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，return_type 是关键字 **void**。
- 函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
- 参数：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
- 函数主体：函数主体包含一组定义函数执行任务的语句。

实例

以下是 **max()** 函数的源代码。该函数有两个参数 num1 和 num2，会返回这两个数中较大的那个数：

```
// 函数返回两个数中较大的那个数

int max(int num1, int num2)
{
   // 局部变量声明
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

函数声明

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

```
return_type function_name( parameter list );
```

针对上面定义的函数 max()，以下是函数声明：

```
int max(int num1, int num2);
```

在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：

```
int max(int, int);
```

当您在一个源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

调用函数

创建 C++ 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。

当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。例如：

```
#include <iostream>
using namespace std;

// 函数声明
int max(int num1, int num2);

int main ()
{
   // 局部变量声明
   int a = 100;
   int b = 200;
   int ret;

   // 调用函数来获取最大值
   ret = max(a, b);

   cout << "Max value is : " << ret << endl;

   return 0;
}

// 函数返回两个数中较大的那个数
int max(int num1, int num2)
{
   // 局部变量声明
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

把 max() 函数和 main() 函数放一块，编译源代码。当运行最后的可执行文件时，会产生下列结果：

```
Max value is : 200
```

函数参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数。

形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有两种向函数传递参数的方式：

调用类型	描述
传值调用	该方法把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数对实际参数没有影响。
指针调用	该方法把参数的地址复制给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。
引用调用	该方法把参数的引用复制给形式参数。在函数内，该引用用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

默认情况下，C++ 使用传值调用来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的参数。之前提到的实例，调用 max() 函数时，使用了相同的方法。

参数的默认值

当您定义一个函数，您可以为参数列表中后边的每一个参数指定默认值。当调用函数时，如果实际参数的值留空，则使用这个默认值。

这是通过在函数定义中使用赋值运算符来为参数赋值的。调用函数时，如果未传递参数的值，则会使用默认值，如果指定了值，则会忽略默认值，使用传递的值。请看下面的实例：

```
#include <iostream>
using namespace std;

int sum(int a, int b=20)
{
  int result;

  result = a + b;

  return (result);
}

int main ()
{
   // 局部变量声明
   int a = 100;
   int b = 200;
   int result;

   // 调用函数来添加值
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // 再次调用函数
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total value is :300
Total value is :120
```

C++ 数字

通常，当我们需要用到数字时，我们会使用原始的数据类型，如 int、short、long、float 和 double 等等。这些用于数字的数据类型，其可能的值和数值范围，我们已经在 C++ 数据类型一章中讨论过。

C++ 定义数字

我们已经在之前章节的各种实例中定义过数字。下面是一个 C++ 中定义各种类型数字的综合实例：

```
#include <iostream>
using namespace std;

int main ()
{
   // 数字定义
   short  s;
   int    i;
   long   l;
   float  f;
   double d;

   // 数字赋值
   s = 10;
   i = 1000;
   l = 1000000;
   f = 230.47;
   d = 30949.374;

   // 数字输出
   cout << "short  s :" << s << endl;
   cout << "int    i :" << i << endl;
   cout << "long   l :" << l << endl;
   cout << "float  f :" << f << endl;
   cout << "double d :" << d << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
short  s :10
int    i :1000
long   l :1000000
float  f :230.47
double d :30949.4
```

C++ 数学运算

在 C++ 中，除了可以创建各种函数，还包含了各种有用的函数供您使用。这些函数写在标准 C 和 C++ 库中，叫做内置函数。您可以在程序中引用这些函数。

C++ 内置了丰富的数学函数，可对各种数字进行运算。下表列出了 C++ 中一些有用的内置的数学函数。

为了利用这些函数，您需要引用数学头文件 。

序号	函数 & 描述
1	**double cos(double);** 该函数返回弧度角（double 型）的余弦。
2	**double sin(double);** 该函数返回弧度角（double 型）的正弦。
3	**double tan(double);** 该函数返回弧度角（double 型）的正切。
4	**double log(double);** 该函数返回参数的自然对数。
5	**double pow(double, double);** 假设第一个参数为 x，第二个参数为 y，则该函数返回 x 的 y 次方。
6	**double hypot(double, double);** 该函数返回两个参数的平方总和的平方根，也就是说，参数为一个直角三角形的两个直角边，函数会返回斜边的长度。
7	**double sqrt(double);** 该函数返回参数的平方根。
8	**int abs(int);** 该函数返回整数的绝对值。
9	**double fabs(double);** 该函数返回任意一个十进制数的绝对值。
10	**double floor(double);** 该函数返回一个小于或等于传入参数的最大整数。

下面是一个关于数学运算的简单实例：

```
#include <iostream>
#include <cmath>
using namespace std;

int main ()
{
   // 数字定义
   short  s = 10;
   int    i = -1000;
   long   l = 100000;
   float  f = 230.47;
   double d = 200.374;

   // 数学运算
   cout << "sin(d) :" << sin(d) << endl;
   cout << "abs(i)  :" << abs(i) << endl;
   cout << "floor(d) :" << floor(d) << endl;
   cout << "sqrt(f) :" << sqrt(f) << endl;
   cout << "pow( d, 2) :" << pow(d, 2) << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
sign(d) :-0.634939
abs(i)  :1000
floor(d) :200
sqrt(f) :15.1812
pow( d, 2 ) :40149.7
```

C++ 随机数

在许多情况下，需要生成随机数。关于随机数生成器，有两个相关的函数。一个是 **rand()**，该函数只返回一个伪随机数。生成随机数之前必须先调用 **srand()** 函数。

下面是一个关于生成随机数的简单实例。实例中使用了 **time()** 函数来获取系统时间的秒数，通过调用 rand() 函数来生成随机数：

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

int main ()
{
   int i,j;

   // 设置种子
   srand( (unsigned)time( NULL ) );

   /* 生成 10 个随机数 */
   for( i = 0; i < 10; i++ )
   {
      // 生成实际的随机数
      j= rand();
      cout <<"随机数： " << j << endl;
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
随机数： 1748144778
随机数： 630873888
随机数： 2134540646
随机数： 219404170
随机数： 902129458
随机数： 920445370
随机数： 1319072661
随机数： 257938873
随机数： 1256201101
随机数： 580322989
```

C++ 数组

C++ 支持数组数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。

数组的声明并不是声明一个个单独的变量，比如 number0、number1、...、number99，而是声明一个数组变量，比如 numbers，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来代表一个个单独的变量。数组中的特定元素可以通过索引访问。

所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素。

声明数组

在 C++ 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示：

```
type arrayName [ arraySize ];
```

这叫做一维数组。**arraySize** 必须是一个大于零的整数常量，**type** 可以是任意有效的 C++ 数据类型。例如，要声明一个类型为 double 的包含 10 个元素的数组 **balance**，声明语句如下：

```
double balance[10];
```

现在 *balance* 是一个可用的数组，可以容纳 10 个类型为 double 的数字。

初始化数组

在 C++ 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

大括号 { } 之间的值的数目不能大于我们在数组声明时在方括号 [ ] 中指定的元素数目。

如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数。因此，如果：

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

您将创建一个数组，它与前一个实例中所创建的数组是完全相同的。下面是一个为数组中某个元素赋值的实例：

```
balance[4] = 50.0;
```

上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，数组的最后一个索引是数组的总大小减去 1。以下是上面所讨论的数组的的图形表示：

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

访问数组元素

数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内，跟在数组名称的后边。例如：

```
double salary = balance[9];
```

上面的语句将把数组中第 10 个元素的值赋给 salary 变量。下面的实例使用了上述的三个概念，即，声明数组、数组赋值、访问数组：

```
#include <iostream>
using namespace std;

#include <iomanip>
using std::setw;

int main ()
{
   int n[ 10 ]; // n 是一个包含 10 个整数的数组

   // 初始化数组元素
   for ( int i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; // 设置元素 i 为 i + 100
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;

   // 输出数组中每个元素的值
   for ( int j = 0; j < 10; j++ )
   {
      cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }

   return 0;
}
```

上面的程序使用了 **setw()** 函数来格式化输出。当上面的代码被编译和执行时，它会产生下列结果：

```
Element        Value
      0          100
      1          101
      2          102
      3          103
      4          104
      5          105
      6          106
      7          107
      8          108
      9          109
```

C++ 中数组详解

在 C++ 中，数组是非常重要的，我们需要了解更多有关数组的细节。下面列出了 C++ 程序员必须清楚的一些与数组相关的重要概念：

概念	描述
多维数组	C++ 支持多维数组。多维数组最简单的形式是二维数组。
指向数组的指针	您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。
传递数组给函数	您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
从函数返回数组	C++ 允许从函数返回数组。

C++ 多维数组

C++ 支持多维数组。多维数组声明的一般形式如下：

```
type name[size1][size2]...[sizeN];
```

例如，下面的声明创建了一个三维 5 . 10 . 4 整型数组：

```
int threedim[5][10][4];
```

二维数组

多维数组最简单的形式是二维数组。一个二维数组，在本质上，是一个一维数组的列表。声明一个 x 行 y 列的二维整型数组，形式如下：

```
type arrayName [ x ][ y ];
```

其中，**type** 可以是任意有效的 C++ 数据类型，**arrayName** 是一个有效的 C++ 标识符。

一个二维数组可以被认为是一个带有 x 行和 y 列的表格。下面是一个二维数组，包含 3 行和 4 列：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

因此，数组中的每个元素是使用形式为 a[i , j] 的元素名称来标识的，其中 a 是数组名称，i 和 j 是唯一标识 a 中每个元素的下标。

初始化二维数组

多维数组可以通过在括号内为每行指定值来进行初始化。下面是一个带有 3 行 4 列的数组。

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  初始化索引号为 0 的行 */
 {4, 5, 6, 7} ,   /*  初始化索引号为 1 的行 */
 {8, 9, 10, 11}   /*  初始化索引号为 2 的行 */
};
```

内部嵌套的括号是可选的，下面的初始化与上面是等同的：

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

访问二维数组元素

二维数组中的元素是通过使用下标（即数组的行索引和列索引）来访问的。例如：

```
int val = a[2][3];
```

上面的语句将获取数组中第 3 行第 4 个元素。您可以通过上面的示意图来进行验证。让我们来看看下面的程序，我们将使用嵌套循环来处理二维数组：

```
#include <iostream>
using namespace std;

int main ()
{
   // 一个带有 5 行 2 列的数组
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

   // 输出数组中每个元素的值
   for ( int i = 0; i < 5; i++ )
      for ( int j = 0; j < 2; j++ )
      {
         cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

如上所述，您可以创建任意维度的数组，但是一般情况下，我们创建的数组是一维数组和二维数组。

C++ 指向数组的指针

您可以先跳过本章，等了解了 C++ 指针的概念之后，再来学习本章的内容。

如果您对 C++ 指针的概念有所了解，那么就可以开始本章的学习。数组名是一个指向数组中第一个元素的常量指针。因此，在下面的声明中：

```
double balance[50];
```

balance 是一个指向 &balance[0] 的指针，即数组 balance 的第一个元素的地址。因此，下面的程序片段把 **p** 赋值为 **balance** 的第一个元素的地址：

```
double *p;
double balance[10];

p = balance;
```

使用数组名作为常量指针是合法的，反之亦然。因此，*(balance + 4) 是一种访问 balance[4] 数据的合法方式。

一旦您把第一个元素的地址存储在 p 中，您就可以使用 *p*、(p+1)、*(p+2) 等来访问数组元素。下面的实例演示了上面讨论到的这些概念：

```
#include <iostream>
using namespace std;

int main ()
{
   // 带有 5 个元素的整型数组
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;

   p = balance;

   // 输出数组中每个元素的值
   cout << "使用指针的数组值 " << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "*(p + " << i << ") : ";
       cout << *(p + i) << endl;
   }

   cout << "使用 balance 作为地址的数组值 " << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "*(balance + " << i << ") : ";
       cout << *(balance + i) << endl;
   }

   return 0;
}



#include <stdio.h>

int main ()
{
   /* 带有 5 个元素的整型数组 */
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;
   int i;

   p = balance;

   /* 输出数组中每个元素的值 */
   printf( "使用指针的数组值\n");
   for ( i = 0; i < 5; i++ )
   {
       printf("*(p + %d) : %f\n",  i, *(p + i) );
   }

   printf( "使用 balance 作为地址的数组值\n");
   for ( i = 0; i < 5; i++ )
   {
       printf("*(balance + %d) : %f\n",  i, *(balance + i) );
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
使用指针的数组值
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
使用 balance 作为地址的数组值
*(balance + 0) : 1000
*(balance + 1) : 2
*(balance + 2) : 3.4
*(balance + 3) : 17
*(balance + 4) : 50
```

在上面的实例中，p 是一个指向 double 型的指针，这意味着它可以存储一个 double 类型的变量。一旦我们有了 p 中的地址，***p** 将给出存储在 p 中相应地址的值，正如上面实例中所演示的。

C++ 传递数组给函数

C++ 不允许向函数传递一个完整的数组作为参数，但是，您可以通过指定不带索引的数组名来传递一个指向数组的指针。

如果您想要在函数中传递一个一维数组作为参数，您必须以下面三种方式来声明函数形式参数，这三种声明方式的结果是一样的，因为每种方式都会告诉编译器将要接收一个整型指针。同样地，您也可以传递一个多维数组作为形式参数。

方式 1

形式参数是一个指针：

```
void myFunction(int *param)
{
.
.
.
}
```

方式 2

形式参数是一个已定义大小的数组：

```
void myFunction(int param[10])
{
.
.
.
}
```

方式 3

形式参数是一个未定义大小的数组：

```
void myFunction(int param[])
{
.
.
.
}
```

实例

现在，让我们来看下面这个函数，它把数组作为参数，同时还传递了另一个参数，根据所传的参数，会返回数组中各元素的平均值：

```
double getAverage(int arr[], int size)
{
  int    i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
   }

  avg = double(sum) / size;

  return avg;
}
```

现在，让我们调用上面的函数，如下所示：

```
#include <iostream>
using namespace std;

// 函数声明
double getAverage(int arr[], int size);

int main ()
{
   // 带有 5 个元素的整型数组
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // 传递一个指向数组的指针作为参数
   avg = getAverage( balance, 5 ) ;

   // 输出返回值
   cout << "平均值是：" << avg << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
平均值是： 214.4
```

您可以看到，就函数而言，数组的长度是无关紧要的，因为 C++ 不会对形式参数执行边界检查。

C++ 从函数返回数组

C++ 不允许返回一个完整的数组作为函数的参数。但是，您可以通过指定不带索引的数组名来返回一个指向数组的指针。

如果您想要从函数返回一个一维数组，您必须声明一个返回指针的函数，如下：

```
int * myFunction()
{
.
.
.
}
```

另外，C++ 不支持在函数外返回局部变量的地址，除非定义局部变量为 **static** 变量。

现在，让我们来看下面的函数，它会生成 10 个随机数，并使用数组来返回它们，具体如下：

```
#include <iostream>
#include <ctime>

using namespace std;

// 要生成和返回随机数的函数
int * getRandom( )
{
  static int  r[10];

  // 设置种子
  srand( (unsigned)time( NULL ) );
  for (int i = 0; i < 10; ++i)
  {
    r[i] = rand();
    cout << r[i] << endl;
  }

  return r;
}

// 要调用上面定义函数的主函数
int main ()
{
   // 一个指向整数的指针
   int *p;

   p = getRandom();
   for ( int i = 0; i < 10; i++ )
   {
       cout << "*(p + " << i << ") : ";
       cout << *(p + i) << endl;
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

C++ 字符串

C++ 提供了以下两种类型的字符串表示形式：

- C 风格字符串
- C++ 引入的 string 类类型

C 风格字符串

C 风格的字符串起源于 C 语言，并在 C++ 中继续得到支持。字符串实际上是使用 **null** 字符 '\0' 终止的一维字符数组。因此，一个以 null 结尾的字符串，包含了组成字符串的字符。

下面的声明和初始化创建了一个 "Hello" 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 "Hello" 的字符数多一个。

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

依据数组初始化规则，您可以把上面的语句写成以下语句：

```
char greeting[] = "Hello";
```

以下是 C/C++ 中定义的字符串的内存表示：

alt="C/C++ 中的字符串表示" src="/wp-content/uploads/2014/08/string_representation.jpg" />
其实，您不需要把 *null* 字符放在字符串常量的末尾。C++ 编译器会在初始化数组时，自动把 '\0' 放在字符串的末尾。让我们尝试输出上面的字符串：

```
#include <iostream>

using namespace std;

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Greeting message: Hello
```

C++ 中有大量的函数用来操作以 null 结尾的字符串：supports a wide range of functions that manipulate null-terminated strings:

序号	函数 & 目的
1	**strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。
2	**strcat(s1, s2);** 连接字符串 s2 到字符串 s1 的末尾。
3	**strlen(s1);** 返回字符串 s1 的长度。
4	**strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回小于 0；如果 s1>s2 则返回大于 0。
5	**strchr(s1, ch);** 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。
6	**strstr(s1, s2);** 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

下面的实例使用了上述的一些函数：

```
#include <iostream>
#include <cstring>

using namespace std;

int main ()
{
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

   // 复制 str1 到 str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // 连接 str1 和 str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // 连接后，str1 的总长度
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

C++ 中的 String 类

C++ 标准库提供了 **string** 类类型，支持上述所有的操作，另外还增加了其他更多的功能。我们将学习 C++ 标准库中的这个类，现在让我们先来看看下面这个实例：

现在您可能还无法透彻地理解这个实例，因为到目前为止我们还没有讨论类和对象。所以现在您可以只是粗略地看下这个实例，等理解了面向对象的概念之后再回头来理解这个实例。

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   // 复制 str1 到 str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // 连接 str1 和 str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // 连接后，str3 的总长度
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

C++ 指针

学习 C++ 的指针既简单又有趣。通过指针，可以简化一些 C++ 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。所以，想要成为一名优秀的 C++ 程序员，学习指针是很有必要的。

正如您所知道的，每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。请看下面的实例，它将输出定义的变量地址：

```
#include <iostream>

using namespace std;

int main ()
{
   int  var1;
   char var2[10];

   cout << "var1 变量的地址： ";
   cout << &var1 << endl;

   cout << "var2 变量的地址： ";
   cout << &var2 << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
var1 变量的地址： 0xbfebd5c0
var2 变量的地址： 0xbfebd5b6
```

通过上面的实例，我们了解了什么是内存地址以及如何访问它。接下来让我们看看什么是指针。

什么是指针？

指针是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

```
type *var-name;
```

在这里，**type** 是指针的基类型，它必须是一个有效的 C++ 数据类型，**var-name** 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

```
int    *ip;    /* 一个整型的指针 */
double *dp;    /* 一个 double 型的指针 */
float  *fp;    /* 一个浮点型的指针 */
char   *ch     /* 一个字符型的指针 */
```

所有指针的值的实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同。

C++ 中使用指针

使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值。下面的实例涉及到了这些操作：

```
#include <iostream>

using namespace std;

int main ()
{
   int  var = 20;   // 实际变量的声明
   int  *ip;        // 指针变量的声明

   ip = &var;       // 在指针变量中存储 var 的地址

   cout << "Value of var variable: ";
   cout << var << endl;

   // 输出在指针变量中存储的地址
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // 访问指针中地址的值
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

C++ 指针详解

在 C++ 中，有很多指针相关的概念，这些概念都很简单，但是都很重要。下面列出了 C++ 程序员必须清楚的一些与指针相关的重要概念：

概念	描述
C++ Null 指针	C++ 支持空指针。NULL 指针是一个定义在标准库中的值为零的常量。
C++ 指针的算术运算	可以对指针进行四种算术运算：++、--、+、-
C++ 指针 vs 数组	指针和数组之间有着密切的关系。
C++ 指针数组	可以定义用来存储指针的数组。
C++ 指向指针的指针	C++ 允许指向指针的指针。
C++ 传递指针给函数	通过引用或地址传递参数，使传递的参数在调用函数中被改变。
C++ 从函数返回指针	C++ 允许函数返回指针到局部变量、静态变量和动态内存分配。

C++ Null 指针

在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值是一个良好的编程习惯。赋为 NULL 值的指针被称为空指针。

NULL 指针是一个定义在标准库中的值为零的常量。请看下面的程序：

```
#include <iostream>

using namespace std;

int main ()
{
   int  *ptr = NULL;

   cout << "ptr 的值是 " << ptr ;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
ptr 的值是 0
```

在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，它表明该指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

如需检查一个空指针，您可以使用 if 语句，如下所示：

```
if(ptr)     /* 如果 p 非空，则完成 */
if(!ptr)    /* 如果 p 为空，则完成 */
```

因此，如果所有未使用的指针都被赋予空值，同时避免使用空指针，就可以防止误用一个未初始化的指针。很多时候，未初始化的变量存有一些垃圾值，导致程序难以调试。

C++ 指针的算术运算

指针是一个用数值表示的地址。因此，您可以对指针执行算术运算。可以对指针进行四种算术运算：++、--、+、-。

假设 **ptr** 是一个指向地址 1000 的整型指针，是一个 32 位的整数，让我们对该指针执行下列的算术运算：

```
ptr++
```

在执行完上述的运算之后，**ptr** 将指向位置 1004，因为 ptr 每增加一次，它都将指向下一个整数位置，即当前位置往后移 4 个字节。这个运算会在不影响内存位置中实际值的情况下，移动指针到下一个内存位置。如果 **ptr** 指向一个地址为 1000 的字符，上面的运算会导致指针指向位置 1001，因为下一个字符位置是在 1001。

递增一个指针

我们喜欢在程序中使用指针代替数组，因为变量指针可以递增，而数组不能递增，因为数组是一个常量指针。下面的程序递增变量指针，以便顺序访问数组中的每一个元素：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // 指针中的数组地址
   ptr = var;
   for (int i = 0; i < MAX; i++)
   {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // 移动到下一个位置
      ptr++;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

递减一个指针

同样地，对指针进行递减运算，即把值减去其数据类型的字节数，如下所示：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // 指针中最后一个元素的地址
   ptr = &var[MAX-1];
   for (int i = MAX; i > 0; i--)
   {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // 移动到下一个位置
      ptr--;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var[3] = 0xbfdb70f8
Value of var[3] = 200
Address of var[2] = 0xbfdb70f4
Value of var[2] = 100
Address of var[1] = 0xbfdb70f0
Value of var[1] = 10
```

指针的比较

指针可以用关系运算符进行比较，如 ==、< 和 >。如果 p1 和 p2 指向两个相关的变量，比如同一个数组中的不同元素，则可对 p1 和 p2 进行大小比较。

下面的程序修改了上面的实例，只要变量指针所指向的地址小于或等于数组的最后一个元素的地址 &var[MAX - 1]，则把变量指针进行递增：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // 指针中第一个元素的地址
   ptr = var;
   int i = 0;
   while ( ptr <= &var[MAX - 1] )
   {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // 指向上一个位置
      ptr++;
      i++;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var[0] = 0xbfce42d0
Value of var[0] = 10
Address of var[1] = 0xbfce42d4
Value of var[1] = 100
Address of var[2] = 0xbfce42d8
Value of var[2] = 200
```

C++ 指针 vs 数组

指针和数组是密切相关的。事实上，指针和数组在很多情况下是可以互换的。例如，一个指向数组开头的指针，可以通过使用指针的算术运算或数组索引来访问数组。请看下面的程序：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // 指针中的数组地址
   ptr = var;
   for (int i = 0; i < MAX; i++)
   {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // 移动到下一个位置
      ptr++;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

然而，指针和数组并不是完全互换的。例如，请看下面的程序：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++)
   {
      *var = i;    // 这是正确的语法
      var++;       // 这是不正确的
   }
   return 0;
}
```

把指针运算符 * 应用到 var 上是完全可以接受的，但修改 var 的值是非法的。这是因为 var 是一个指向数组开头的常量，不能作为左值。

由于一个数组名对应一个指针常量，只要不改变数组的值，仍然可以用指针形式的表达式。例如，下面是一个有效的语句，把 var[2] 赋值为 500：

```
*(var + 2) = 500;
```

上面的语句是有效的，且能成功编译，因为 var 未改变。

C++ 指针数组

在我们讲解指针数组的概念之前，先让我们来看一个实例，它用到了一个由 3 个整数组成的数组：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++)
   {
      cout << "Value of var[" << i << "] = ";
      cout << var[i] << endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

可能有一种情况，我们想要让数组存储指向 int 或 char 或其他数据类型的指针。下面是一个指向整数的指针数组的声明：

```
int *ptr[MAX];
```

在这里，把 **ptr** 声明为一个数组，由 MAX 个整数指针组成。因此，ptr 中的每个元素，都是一个指向 int 值的指针。下面的实例用到了三个整数，它们将存储在一个指针数组中，如下所示：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int *ptr[MAX];

   for (int i = 0; i < MAX; i++)
   {
      ptr[i] = &var[i]; // 赋值为整数的地址
   }
   for (int i = 0; i < MAX; i++)
   {
      cout << "Value of var[" << i << "] = ";
      cout << *ptr[i] << endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

您也可以用一个指向字符的指针数组来存储一个字符串列表，如下：

```
#include <iostream>

using namespace std;
const int MAX = 4;

int main ()
{
   char *names[MAX] = {
                   "Zara Ali",
                   "Hina Ali",
                   "Nuha Ali",
                   "Sara Ali",
   };

   for (int i = 0; i < MAX; i++)
   {
      cout << "Value of names[" << i << "] = ";
      cout << names[i] << endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

C++ 指向指针的指针（多级间接寻址）

指向指针的指针是一种多级间接寻址的形式，或者说是一个指针链。通常，一个指针包含一个变量的地址。当我们定义一个指向指针的指针时，第一个指针包含了第二个指针的地址，第二个指针指向包含实际值的位置。

一个指向指针的指针变量必须如下声明，即在变量名前放置两个星号。例如，下面声明了一个指向 int 类型指针的指针：

```
int **var;
```

当一个目标值被一个指针间接指向到另一个指针时，访问这个值需要使用两个星号运算符，如下面实例所示：

```
#include <iostream>

using namespace std;

int main ()
{
   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   // 获取 var 的地址
   ptr = &var;

   // 使用运算符 & 获取 ptr 的地址
   pptr = &ptr;

   // 使用 pptr 获取值
   cout << "Value of var :" << var << endl;
   cout << "Value available at *ptr :" << *ptr << endl;
   cout << "Value available at **pptr :" << **pptr << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

C++ 传递指针给函数

C++ 允许您传递指针给函数，只需要简单地声明函数参数为指针类型即可。

下面的实例中，我们传递一个无符号的 long 型指针给函数，并在函数内改变这个值：

```
#include <iostream>
#include <ctime>

using namespace std;
void getSeconds(unsigned long *par);

int main ()
{
   unsigned long sec;


   getSeconds( &sec );

   // 输出实际值
   cout << "Number of seconds :" << sec << endl;

   return 0;
}

void getSeconds(unsigned long *par)
{
   // 获取当前的秒数
   *par = time( NULL );
   return;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Number of seconds :1294450468
```

能接受指针作为参数的函数，也能接受数组作为参数，如下所示：

```
#include <iostream>
using namespace std;

// 函数声明
double getAverage(int *arr, int size);

int main ()
{
   // 带有 5 个元素的整型数组
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // 传递一个指向数组的指针作为参数
   avg = getAverage( balance, 5 ) ;

   // 输出返回值
   cout << "Average value is: " << avg << endl;

   return 0;
}

double getAverage(int *arr, int size)
{
  int    i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
   }

  avg = double(sum) / size;

  return avg;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Average value is: 214.4
```

C++ 从函数返回指针

在上一章中，我们已经了解了 C++ 中如何从函数返回数组，类似地，C++ 允许您从函数返回指针。为了做到这点，您必须声明一个返回指针的函数，如下所示：

```
int * myFunction()
{
.
.
.
}
```

另外，C++ 不支持在函数外返回局部变量的地址，除非定义局部变量为 **static** 变量。

现在，让我们来看下面的函数，它会生成 10 个随机数，并使用表示指针的数组名（即第一个数组元素的地址）来返回它们，具体如下：

```
#include <iostream>
#include <ctime>

using namespace std;

// 要生成和返回随机数的函数
int * getRandom( )
{
  static int  r[10];

  // 设置种子
  srand( (unsigned)time( NULL ) );
  for (int i = 0; i < 10; ++i)
  {
    r[i] = rand();
    cout << r[i] << endl;
  }

  return r;
}

// 要调用上面定义函数的主函数
int main ()
{
   // 一个指向整数的指针
   int *p;

   p = getRandom();
   for ( int i = 0; i < 10; i++ )
   {
       cout << "*(p + " << i << ") : ";
       cout << *(p + i) << endl;
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

C++ 引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。

C++ 引用 vs 指针

引用很容易与指针混淆，它们之间有三个主要的不同：

- 不存在空引用。引用必须连接到一块合法的内存。
- 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
- 引用必须在创建时被初始化。指针可以在任何时间被初始化。

C++ 中创建引用

试想变量名称是变量附属在内存位置中的标签，您可以把引用当成是变量附属在内存位置中的第二个标签。因此，您可以通过原始变量名称或引用来访问变量的内容。例如：

```
int    i = 17;
```

我们可以为 i 声明引用变量，如下所示：

```
int&amp;    r = i;
```

在这些声明中，& 读作引用。因此，第一个声明可以读作 "r 是一个初始化为 i 的整型引用"，第二个声明可以读作 "s 是一个初始化为 d 的 double 型引用"。下面的实例使用了 int 和 double 引用：

```
#include <iostream>

using namespace std;

int main ()
{
   // 声明简单的变量
   int    i;
   double d;

   // 声明引用变量
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

引用通常用于函数参数列表和函数返回值。下面列出了 C++ 程序员必须清楚的两个与 C++ 引用相关的重要概念：

概念	描述
把引用作为参数	C++ 支持把引用作为参数传给函数，这比传一般的参数更安全。
把引用作为返回值	可以从 C++ 函数中返回引用，就像返回其他数据类型一样。

C++ 把引用作为参数

我们已经讨论了如何使用指针来实现引用调用函数。下面的实例使用了引用来实现引用调用函数。

```
#include <iostream>
using namespace std;

// 函数声明
void swap(int& x, int& y);

int main ()
{
   // 局部变量声明
   int a = 100;
   int b = 200;

   cout << "交换前，a 的值：" << a << endl;
   cout << "交换前，b 的值：" << b << endl;

   /* 调用函数来交换值 */
   swap(a, b);

   cout << "交换后，a 的值：" << a << endl;
   cout << "交换前，b 的值：" << b << endl;

   return 0;
}

// 函数定义
void swap(int& x, int& y)
{
   int temp;
   temp = x; /* 保存地址 x 的值 */
   x = y;    /* 把 y 赋值给 x */
   y = temp; /* 把 x 赋值给 y  */

   return;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
交换前，a 的值： 100
交换前，b 的值： 200
交换后，a 的值： 200
交换后，b 的值： 100
```

C++ 把引用作为返回值

通过使用引用来替代指针，会使 C++ 程序更容易阅读和维护。C++ 函数可以返回一个引用，方式与返回一个指针类似。

当函数返回一个引用时，则返回一个指向返回值的隐式指针。这样，函数就可以放在赋值语句的左边。例如，请看下面这个简单的程序：

```
#include <iostream>
#include <ctime>

using namespace std;

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues( int i )
{
  return vals[i];   // 返回第 i 个元素的引用
}

// 要调用上面定义函数的主函数
int main ()
{

   cout << "改变前的值" << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "vals[" << i << "] = ";
       cout << vals[i] << endl;
   }

   setValues(1) = 20.23; // 改变第 2 个元素
   setValues(3) = 70.8;  // 改变第 4 个元素

   cout << "改变后的值" << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "vals[" << i << "] = ";
       cout << vals[i] << endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
改变前的值
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
vals[3] = 24.1
vals[4] = 50
改变后的值
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
vals[4] = 50
```

当返回一个引用时，要注意被引用的对象不能超出作用域。所以返回一个对局部变量的引用是不合法的，但是，可以返回一个对静态变量的引用。

```
int& func() {
   int q;
   //! return q; // 在编译时发生错误
   static int x;
   return x;     // 安全，x 在函数作用域外依然是有效的
}
```

C++ 日期 & 时间

C++ 标准库没有提供所谓的日期类型。C++ 继承了 C 语言用于日期和时间操作的结构和函数。为了使用日期和时间相关的函数和结构，需要在 C++ 程序中引用 头文件。

有四个与时间相关的类型：**clock_t**、**time_t**、**size_t** 和 **tm**。类型 clock_t、size_t 和 time_t 能够把系统时间和日期表示为某种整数。

结构类型 **tm** 把日期和时间以 C 结构的形式保存，tm 结构的定义如下：

```
struct tm {
  int tm_sec;   // 秒，正常范围从 0 到 59，但允许至 61
  int tm_min;   // 分，范围从 0 到 59
  int tm_hour;  // 小时，范围从 0 到 23
  int tm_mday;  // 一月中的第几天，范围从 1 到 31
  int tm_mon;   // 月，范围从 0 到 11
  int tm_year;  // 自 1900 年起的年数
  int tm_wday;  // 一周中的第几天，范围从 0 到 6，从星期日算起
  int tm_yday;  // 一年中的第几天，范围从 0 到 365，从 1 月 1 日算起
  int tm_isdst; // 夏令时
}
```

下面是 C/C++ 中关于日期和时间的重要函数。所有这些函数都是 C/C++ 标准库的组成部分，您可以在 C++ 标准库中查看一下各个函数的细节。

序号	函数 & 描述
1	**time_t time(time_t *time);** 该函数返回系统的当前日历时间，自 1970 年 1 月 1 日以来经过的秒数。如果系统没有时间，则返回 .1。
2	**char *ctime(const time_t *time);** 该返回一个表示当地时间的字符串指针，字符串形式 *day month year hours:minutes:seconds year\n\0*。
3	**struct tm *localtime(const time_t *time);** 该函数返回一个指向表示本地时间的 **tm** 结构的指针。
4	**clock_t clock(void);** 该函数返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。如果时间不可用，则返回 .1。
5	**char * asctime (const struct tm * time);** 该函数返回一个指向字符串的指针，字符串包含了 time 所指向结构中存储的信息，返回形式为：day month date hours:minutes:seconds year\n\0。
6	**struct tm *gmtime(const time_t *time);** 该函数返回一个指向 time 的指针，time 为 tm 结构，用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。
7	**time_t mktime(struct tm *time);** 该函数返回日历时间，相当于 time 所指向结构中存储的时间。
8	**double difftime (time_t time2, time_t time1);** 该函数返回 time1 和 time2 之间相差的秒数。
9	**size_t strftime();** 该函数可用于格式化日期和时间为指定的格式。

当前日期和时间

下面的实例获取当前系统的日期和时间，包括本地时间和协调世界时（UTC）。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
   // 基于当前系统的当前日期/时间
   time_t now = time(0);

   // 把 now 转换为字符串形式
   char* dt = ctime(&now);

   cout << "本地日期和时间：" << dt << endl;

   // 把 now 转换为 tm 结构
   tm *gmtm = gmtime(&now);
   dt = asctime(gmtm);
   cout << "UTC 日期和时间："<< dt << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
本地日期和时间：Sat Jan  8 20:07:41 2011

UTC 日期和时间：Sun Jan  9 03:07:41 2011
```

使用结构 tm 格式化时间

tm 结构在 C/C++ 中处理日期和时间相关的操作时，显得尤为重要。tm 结构以 C 结构的形式保存日期和时间。大多数与时间相关的函数都使用了 tm 结构。下面的实例使用了 tm 结构和各种与日期和时间相关的函数。

在练习使用结构之前，需要对 C 结构有基本的了解，并懂得如何使用箭头 -> 运算符来访问结构成员。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
   // 基于当前系统的当前日期/时间
   time_t now = time(0);

   cout << "Number of sec since January 1,1970:" << now << endl;

   tm *ltm = localtime(&now);

   // 输出 tm 结构的各个组成部分
   cout << "Year: "<< 1900 + ltm->tm_year << endl;
   cout << "Month: "<< 1 + ltm->tm_mon<< endl;
   cout << "Day: "<<  ltm->tm_mday << endl;
   cout << "Time: "<< 1 + ltm->tm_hour << ":";
   cout << 1 + ltm->tm_min << ":";
   cout << 1 + ltm->tm_sec << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Number of sec since January 1, 1970:1294548238
Year: 2011
Month: 1
Day: 8
Time: 22: 44:59
```

C++ 基本的输入输出

C++ 标准库提供了一组丰富的输入/输出功能，我们将在后续的章节进行介绍。本章将讨论 C++ 编程中最基本和最常见的 I/O 操作。

C++ 的 I/O 发生在流中，流是字节序列。如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做输入操作。如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做输出操作。

I/O 库头文件

下列的头文件在 C++ 编程中很重要。

头文件	函数和描述
`<iostream>`	该文件定义了 **cin**、**cout**、**cerr** 和 **clog** 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。
`<iomanip>`	该文件通过所谓的参数化的流操纵器（比如 **setw** 和 **setprecision**），来声明对执行标准化 I/O 有用的服务。
`<fstream>`	该文件为用户控制的文件处理声明服务。我们将在文件和流的相关章节讨论它的细节。

标准输出流（cout）

预定义的对象 **cout** 是 **ostream** 类的一个实例。cout 对象"连接"到标准输出设备，通常是显示屏。**cout** 是与流插入运算符 << 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Hello C++";

   cout << "Value of str is : " << str << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of str is : Hello C++
```

C++ 编译器根据要输出变量的数据类型，选择合适的流插入运算符来显示值。<< 运算符被重载来输出内置类型（整型、浮点型、double 型、字符串和指针）的数据项。

流插入运算符 << 在一个语句中可以多次使用，如上面实例中所示，**endl** 用于在行末添加一个换行符。

标准输入流（cin）

预定义的对象 **cin** 是 **istream** 类的一个实例。cin 对象附属到标准输入设备，通常是键盘。**cin** 是与流提取运算符 >> 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

int main( )
{
   char name[50];

   cout << "请输入您的名称： ";
   cin >> name;
   cout << "您的名称是： " << name << endl;

}
```

当上面的代码被编译和执行时，它会提示用户输入名称。当用户输入一个值，并按回车键，就会看到下列结果：

```
请输入您的名称： cplusplus
您的名称是： cplusplus
```

C++ 编译器根据要输入值的数据类型，选择合适的流提取运算符来提取值，并把它存储在给定的变量中。

流提取运算符 >> 在一个语句中可以多次使用，如果要求输入多个数据，可以使用如下语句：

```
cin >> name >> age;
```

这相当于下面两个语句：

```
cin >> name;
cin >> age;
```

标准错误流（cerr）

预定义的对象 **cerr** 是 **ostream** 类的一个实例。cerr 对象附属到标准错误设备，通常也是显示屏，但是 **cerr** 对象是非缓冲的，且每个流插入到 cerr 都会立即输出。

cerr 也是与流插入运算符 << 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Error message : Unable to read....
```

标准日志流（clog）

预定义的对象 **clog** 是 **ostream** 类的一个实例。clog 对象附属到标准错误设备，通常也是显示屏，但是 **clog** 对象是缓冲的。这意味着每个流插入到 clog 都会先存储在缓冲在，直到缓冲填满或者缓冲区刷新时才会输出。

clog 也是与流插入运算符 << 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

   clog << "Error message : " << str << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Error message : Unable to read....
```

通过这些小实例，我们无法区分 cout、cerr 和 clog 的差异，但在编写和执行大型程序时，它们之间的差异就变得非常明显。所以良好的编程实践告诉我们，使用 cerr 流来显示错误消息，而其他的日志消息则使用 clog 流来输出。

C++ 数据结构

C/C++ 数组允许定义可存储相同类型数据项的变量，但是结构是 C++ 中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。

结构用于表示一条记录，假设您想要跟踪图书馆中书本的动态，您可能需要跟踪每本书的下列属性：

- Title
- Author
- Subject
- Book ID

定义结构

为了定义结构，您必须使用 **struct** 语句。struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下：

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

structure tag 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在结构定义的末尾，最后一个分号之前，您可以指定一个或多个结构变量，这是可选的。下面是声明 Book 结构的方式：

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}book;
```

访问结构成员

为了访问结构的成员，我们使用成员访问运算符（**.**）。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。您可以使用 **struct** 关键字来定义结构类型的变量。下面的实例演示了结构的用法：

```
#include <iostream>
#include <cstring>

using namespace std;

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        // 声明 Book1，类型为 Book
   struct Books Book2;        // 声明 Book2，类型为 Book

   // Book1 详述
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book2 详述
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // 输出 Book1 信息
   cout << "Book 1 title : " << Book1.title <<endl;
   cout << "Book 1 author : " << Book1.author <<endl;
   cout << "Book 1 subject : " << Book1.subject <<endl;
   cout << "Book 1 id : " << Book1.book_id <<endl;

   // 输出 Book2 信息
   cout << "Book 2 title : " << Book2.title <<endl;
   cout << "Book 2 author : " << Book2.author <<endl;
   cout << "Book 2 subject : " << Book2.subject <<endl;
   cout << "Book 2 id : " << Book2.book_id <<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

结构作为函数参数

您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。您可以使用上面实例中的方式来访问结构变量：

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books book );

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        // 声明 Book1，类型为 Book
   struct Books Book2;        // 声明 Book2，类型为 Book

   // Book1 详述
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book2 详述
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // 输出 Book1 信息
   printBook( Book1 );

   // 输出 Book2 信息
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book )
{
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
   cout << "Book id : " << book.book_id <<endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

指向结构的指针

您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

```
struct Books *struct_pointer;
```

现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：

```
struct_pointer = &Book1;
```

为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：

```
struct_pointer->title;
```

让我们使用结构指针来重写上面的实例，这将有助于您理解结构指针的概念：

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books *book );

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        // 声明 Book1，类型为 Book
   struct Books Book2;        // 声明 Book2，类型为 Book */

   // Book1 详述
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book2 详述
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // 通过传 Book1 的地址来输出 Book1 信息
   printBook( &Book1 );

   // 通过传 Book2 的地址来输出 Book2 信息
   printBook( &Book2 );

   return 0;
}
// 该函数以结构指针作为参数
void printBook( struct Books *book )
{
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

typedef 关键字

下面是一种更简单的定义结构的方式，您可以为创建的类型取一个"别名"。例如：

```
typedef struct
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}Books;
```

现在，您可以直接使用 *Books* 来定义 *Books* 类型的变量，而不需要使用 struct 关键字。下面是实例：

```
Books Book1, Book2;
```

您可以使用 **typedef** 关键字来定义非结构类型，如下所示：

```
typedef long int *pint32;

pint32 x, y, z;
```

x, y 和 z 都是指向长整型 long int 的指针。

C++ 面向对象

C++ 类 & 对象

C++ 在 C 语言的基础上增加了面向对象编程，C++ 支持面向对象程序设计。类是 C++ 的核心特性，通常被称为用户定义的类型。

类用于指定对象的形式，它包含了数据表示法和用于处理数据的方法。类中的数据和方法称为类的成员。函数在一个类被称为类的成员。

C++ 类定义

定义一个类，本质上是定义一个数据类型的蓝图。这实际上并没有定义任何数据，但它定义了类的名称意味着什么，也就是说，它定义了类的对象包括了什么，以及可以在这个对象上执行哪些操作。

类定义是以关键字 **class** 开头，后跟类的名称。类的主体是包含在一对花括号中。类定义后必须跟着一个分号或一个声明列表。例如，我们使用关键字 **class** 定义 Box 数据类型，如下所示：

```
class Box
{
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

关键字 **public** 确定了类成员的访问属性。在类对象作用域内，公共成员在类的外部是可访问的。您也可以指定类的成员为 **private** 或 **protected**，这个我们稍后会进行讲解。

定义 C++ 对象

类提供了对象的蓝图，所以基本上，对象是根据类来创建的。声明类的对象，就像声明基本类型的变量一样。下面的语句声明了类 Box 的两个对象：

```
Box Box1;          // 声明 Box1，类型为 Box
Box Box2;          // 声明 Box2，类型为 Box
```

对象 Box1 和 Box2 都有它们各自的数据成员。

访问数据成员

类的对象的公共数据成员可以使用直接成员访问运算符 (.) 来访问。为了更好地理解这些概念，让我们尝试一下下面的实例：

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;   // 长度
      double breadth;  // 宽度
      double height;   // 高度
};

int main( )
{
   Box Box1;        // 声明 Box1，类型为 Box
   Box Box2;        // 声明 Box2，类型为 Box
   double volume = 0.0;     // 用于存储体积

   // box 1 详述
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 详述
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;

   // box 1 的体积
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Box1 的体积：" << volume <<endl;

   // box 2 的体积
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Box2 的体积：" << volume <<endl;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积：210
Box2 的体积：1560
```

需要注意的是，私有的成员和受保护的成员不能使用直接成员访问运算符 (.) 来直接访问。我们将在后续的教程中学习如何访问私有成员和受保护的成员。

类 & 对象详解

到目前为止，我们已经对 C++ 的类和对象有了基本的了解。下面的列表中还列出了其他一些 C++ 类和对象相关的概念，可以点击相应的链接进行学习。

概念	描述
类成员函数	类的成员函数是指那些把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。
类访问修饰符	类成员可以被定义为 public、private 或 protected。默认情况下是定义为 private。
构造函数 & 析构函数	类的构造函数是一种特殊的函数，在创建一个新的对象时调用。类的析构函数也是一种特殊的函数，在删除所创建的对象时调用。
C++ 拷贝构造函数	拷贝构造函数，是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。
C++ 友元函数	友元函数可以访问类的 private 和 protected 成员。
C++ 内联函数	通过内联函数，编译器试图在调用函数的地方扩展函数体中的代码。
C++ 中的 this 指针	每个对象都有一个特殊的指针 **this**，它指向对象本身。
C++ 中指向类的指针	指向类的指针方式如同指向结构的指针。实际上，类可以看成是一个带有函数的结构。
C++ 类的静态成员	类的数据成员和函数成员都可以被声明为静态的。

C++ 类成员函数

类的成员函数是指那些把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。类成员函数是类的一个成员，它可以操作类的任意对象，可以访问对象中的所有成员。

让我们看看之前定义的类 Box，现在我们要使用成员函数来访问类的成员，而不是直接访问这些类的成员：

```
class Box
{
   public:
      double length;         // 长度
      double breadth;        // 宽度
      double height;         // 高度
      double getVolume(void);// 返回体积
};
```

成员函数可以定义在类定义内部，或者单独使用范围解析运算符 **::** 来定义。在类定义中定义的成员函数把函数声明为内联的，即便没有使用 inline 标识符。所以您可以按照如下方式定义 **Volume()** 函数：

```
class Box
{
   public:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度

      double getVolume(void)
      {
         return length * breadth * height;
      }
};
```

您也可以在类的外部使用范围解析运算符 **::** 定义该函数，如下所示：

```
double Box::getVolume(void)
{
    return length * breadth * height;
}
```

在这里，需要强调一点，在 :: 运算符之前必须使用类名。调用成员函数是在对象上使用点运算符（**.**），这样它就能操作与该对象相关的数据，如下所示：

```
Box myBox;          // 创建一个对象

myBox.getVolume();  // 调用该对象的成员函数
```

让我们使用上面提到的概念来设置和获取类中不同的成员的值：

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;         // 长度
      double breadth;        // 宽度
      double height;         // 高度

      // 成员函数声明
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// 成员函数定义
double Box::getVolume(void)
{
    return length * breadth * height;
}

void Box::setLength( double len )
{
    length = len;
}

void Box::setBreadth( double bre )
{
    breadth = bre;
}

void Box::setHeight( double hei )
{
    height = hei;
}

// 程序的主函数
int main( )
{
   Box Box1;                // 声明 Box1，类型为 Box
   Box Box2;                // 声明 Box2，类型为 Box
   double volume = 0.0;     // 用于存储体积

   // box 1 详述
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 详述
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // box 1 的体积
   volume = Box1.getVolume();
   cout << "Box1 的体积：" << volume <<endl;

   // box 2 的体积
   volume = Box2.getVolume();
   cout << "Box2 的体积：" << volume <<endl;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
```

C++ 类访问修饰符

数据隐藏是面向对象编程的一个重要特点，它防止函数直接访问类类型的内部成员。类成员的访问限制是通过在类主体内部对各个区域标记 **public**、**private**、**protected** 来指定的。关键字 public、private、protected 称为访问说明符。

一个类可以有多个 public、protected 或 private 标记区域。每个标记区域在下一个标记区域开始之前或者在遇到类主体结束右括号之前都是有效的。成员和类的默认访问修饰符是 private。

```
class Base {
   public:
  // public members go here
   protected:
  // protected members go here
   private:
  // private members go here
};
```

公有（public）成员

公有成员在程序中类的外部是可访问的。您可以不使用任何成员函数来设置和获取公有变量的值，如下所示：

```
#include <iostream>

using namespace std;

class Line
{
   public:
      double length;
      void setLength( double len );
      double getLength( void );
};

// 成员函数定义
double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}

// 程序的主函数
int main( )
{
   Line line;

   // 设置长度
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // 不使用成员函数设置长度
   line.length = 10.0; // OK: 因为 length 是公有的
   cout << "Length of line : " << line.length <<endl;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Length of line : 6
Length of line : 10
```

私有（private）成员

私有成员变量或函数在类的外部是不可访问的，甚至是不可查看的。只有类和友元函数可以访问私有成员。

默认情况下，类的所有成员都是私有的。例如在下面的类中，**width** 是一个私有成员，这意味着，如果您没有使用任何访问修饰符，类的成员将被假定为私有成员：

```
class Box
{
   double width;
   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );
};
```

实际操作中，我们一般会在私有区域定义数据，在公有区域定义相关的函数，以便在类的外部也可以调用这些函数，如下所示：

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );

   private:
      double width;
};

// 成员函数定义
double Box::getWidth(void)
{
    return width ;
}

void Box::setWidth( double wid )
{
    width = wid;
}

// 程序的主函数
int main( )
{
   Box box;

   // 不使用成员函数设置长度
   box.length = 10.0; // OK: 因为 length 是公有的
   cout << "Length of box : " << box.length <<endl;

   // 不使用成员函数设置宽度
   // box.width = 10.0; // Error: 因为 width 是私有的
   box.setWidth(10.0);  // 使用成员函数设置宽度
   cout << "Width of box : " << box.getWidth() <<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Length of box :  10  Width of box :  10
```

保护（protected）成员

保护成员变量或函数与私有成员十分相似，但有一点不同，保护成员在派生类（即子类）中是可访问的。

在下一个章节中，您将学习到派生类和继承的知识。现在您可以看到下面的实例中，我们从父类 **Box** 派生了一个子类 **smallBox**。

下面的实例与前面的实例类似，在这里 **width** 成员可被派生类 smallBox 的任何成员函数访问。

```
#include <iostream>
using namespace std;

class Box
{
   protected:
      double width;
};

class SmallBox:Box // SmallBox 是派生类
{
   public:
      void setSmallWidth( double wid );
      double getSmallWidth( void );
};

// 子类的成员函数
double SmallBox::getSmallWidth(void)
{
    return width ;
}

void SmallBox::setSmallWidth( double wid )
{
    width = wid;
}

// 程序的主函数
int main( )
{
   SmallBox box;

   // 使用成员函数设置宽度
   box.setSmallWidth(5.0);
   cout << "Width of box : "<< box.getSmallWidth() << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Width of box :  5
```

C++ 类构造函数 & 析构函数

类的构造函数

类的构造函数是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

构造函数的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。构造函数可用于为某些成员变量设置初始值。

下面的实例有助于更好地理解构造函数的概念：

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // 这是构造函数

   private:
      double length;
};

// 成员函数定义，包括构造函数
Line::Line(void)
{
    cout << "Object is being created" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// 程序的主函数
int main( )
{
   Line line;

   // 设置长度
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Object is being created
Length of line : 6
```

带参数的构造函数

默认的构造函数没有任何参数，但如果需要，构造函数也可以带有参数。这样在创建对象时就会给对象赋初始值，如下面的例子所示：

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // 这是构造函数

   private:
      double length;
};

// 成员函数定义，包括构造函数
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// 程序的主函数
int main( )
{
   Line line(10.0);

   // 获取默认设置的长度
   cout << "Length of line : " << line.getLength() <<endl;
   // 再次设置长度
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

使用初始化列表来初始化字段

使用初始化列表来初始化字段：

```
Line::Line( double len): length(len)
{
    cout << "Object is being created, length = " << len << endl;
}
```

上面的语法等同于如下语法：

```
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}
```

假设有一个类 C，具有多个字段 X、Y、Z 等需要进行初始化，同理地，您可以使用上面的语法，只需要在不同的字段使用逗号进行分隔，如下所示：

```
C::C( double a, double b, double c): X(a), Y(b), Z(c)
{
  ....
}
```

类的析构函数

类的析构函数是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行。

析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀，它不会返回任何值，也不能带有任何参数。析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源。

下面的实例有助于更好地理解析构函数的概念：

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // 这是构造函数声明
      ~Line();  // 这是析构函数声明

   private:
      double length;
};

// 成员函数定义，包括构造函数
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// 程序的主函数
int main( )
{
   Line line;

   // 设置长度
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Object is being created
Length of line : 6
Object is being deleted
```

C++ 拷贝构造函数

拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：

- 通过使用另一个同类型的对象来初始化新创建的对象。
- 复制对象把它作为参数传递给函数。
- 复制对象，并从函数返回这个对象。

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。拷贝构造函数的最常见形式如下：

```
classname (const classname &obj) {
   // 构造函数的主体
}
```

在这里，**obj** 是一个对象引用，该对象是用于初始化另一个对象的。

```
#include <iostream>

using namespace std;

class Line
{
   public:
      int getLength( void );
      Line( int len );             // 简单的构造函数
      Line( const Line &obj);  // 拷贝构造函数
      ~Line();                     // 析构函数

   private:
      int *ptr;
};

// 成员函数定义，包括构造函数
Line::Line(int len)
{
    cout << "Normal constructor allocating ptr" << endl;
    // 为指针分配内存
    ptr = new int;
    *ptr = len;
}

Line::Line(const Line &obj)
{
    cout << "Copy constructor allocating ptr." << endl;
    ptr = new int;
   *ptr = *obj.ptr; // copy the value
}

Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}

void display(Line obj)
{
   cout << "Length of line : " << obj.getLength() <<endl;
}

// 程序的主函数
int main( )
{
   Line line(10);

   display(line);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

下面的实例对上面的实例稍作修改，通过使用已有的同类型的对象来初始化新创建的对象：

```
#include <iostream>

using namespace std;

class Line
{
   public:
      int getLength( void );
      Line( int len );             // 简单的构造函数
      Line( const Line &obj);  // 拷贝构造函数
      ~Line();                     // 析构函数

   private:
      int *ptr;
};

// 成员函数定义，包括构造函数
Line::Line(int len)
{
    cout << "Normal constructor allocating ptr" << endl;
    // 为指针分配内存
    ptr = new int;
    *ptr = len;
}

Line::Line(const Line &obj)
{
    cout << "Copy constructor allocating ptr." << endl;
    ptr = new int;
   *ptr = *obj.ptr; // copy the value
}

Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}

void display(Line obj)
{
   cout << "Length of line : " << obj.getLength() <<endl;
}

// 程序的主函数
int main( )
{
   Line line1(10);

   Line line2 = line1; // 这里也调用了拷贝构造函数

   display(line1);
   display(line2);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
Freeing memory!
```

C++ 友元函数

类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

如果要声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 **friend**，如下所示：

```
class Box
{
   double width;
public:
   double length;
   friend void printWidth( Box box );
   void setWidth( double wid );
};
```

声明类 ClassTwo 的所有成员函数作为类 ClassOne 的友元，需要在类 ClassOne 的定义中放置如下声明：

```
friend  class  ClassTwo;
```

请看下面的程序：

```
#include <iostream>

using namespace std;

class Box
{
   double width;
public:
   friend void printWidth( Box box );
   void setWidth( double wid );
};

// 成员函数定义
void Box::setWidth( double wid )
{
    width = wid;
}

// 请注意：printWidth() 不是任何类的成员函数
void printWidth( Box box )
{
   /* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 */
   cout << "Width of box : " << box.width <<endl;
}

// 程序的主函数
int main( )
{
   Box box;

   // 不使用成员函数设置宽度
   box.setWidth(10.0);

   // 使用友元函数输出宽度
   printWidth( box );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Width of box : 10
```

C++ 内联函数

C++ 内联函数是通常与类一起使用。如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。

对内联函数进行任何修改，都需要重新编译函数的所有客户端，因为编译器需要重新更换一次所有的代码，否则将会继续使用旧的函数。

如果想把一个函数定义为内联函数，则需要在函数名前面放置关键字 **inline**，在调用函数之前需要对函数进行定义。如果已定义的函数多于一行，编译器会忽略 inline 限定符。的情况下定义的函数多了一行。

在类定义中的定义的函数都是内联函数，即使没有使用 **inline** 说明符。

下面是一个实例，使用内联函数来返回两个数中的最大值：

```
#include <iostream>

using namespace std;

inline int Max(int x, int y)
{
   return (x > y)? x : y;
}

// 程序的主函数
int main( )
{

   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

C++ this 指针

在 C++ 中，每一个对象都能通过 **this** 指针来访问自己的地址。**this** 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

友元函数没有 **this** 指针，因为友元不是类的成员。只有成员函数才有 **this** 指针。

下面的实例有助于更好地理解 this 指针的概念：

```
#include <iostream>

using namespace std;

class Box
{
   public:
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      int compare(Box box)
      {
         return this->Volume() > box.Volume();
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   if(Box1.compare(Box2))
   {
      cout << "Box2 is smaller than Box1" <<endl;
   }
   else
   {
      cout << "Box2 is equal to or larger than Box1" <<endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

C++ 指向类的指针

一个指向 C++ 类的指针与指向结构的指针类似，访问指向类的指针的成员，需要使用成员访问运算符 **->**，就像访问指向结构的指针一样。与所有的指针一样，您必须在使用指针之前，对指针进行初始化。

下面的实例有助于更好地理解指向类的指针的概念：

```
#include <iostream>

using namespace std;

class Box
{
   public:
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   Box *ptrBox;                // Declare pointer to a class.

   // 保存第一个对象的地址
   ptrBox = &Box1;

   // 现在尝试使用成员访问运算符来访问成员
   cout << "Volume of Box1: " << ptrBox->Volume() << endl;

   // 保存第二个对象的地址
   ptrBox = &Box2;

   // 现在尝试使用成员访问运算符来访问成员
   cout << "Volume of Box2: " << ptrBox->Volume() << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

C++ 类的静态成员

我们可以使用 **static** 关键字来把类成员定义为静态的。当我们声明类的成员为静态时，这意味着无论创建多少个类的对象，静态成员都只有一个副本。

静态成员在类的所有对象中是共享的。如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。我们不能把静态成员放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 **::** 来重新声明静态变量从而对它进行初始化，如下面的实例所示。

下面的实例有助于更好地理解静态数据成员的概念：

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // 每次创建对象时增加 1
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // 长度
      double breadth;    // 宽度
      double height;     // 高度
};

// 初始化类 Box 的静态成员
int Box::objectCount = 0;

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // 声明 box1
   Box Box2(8.5, 6.0, 2.0);    // 声明 box2

   // 输出对象的总数
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Constructor called.
Constructor called.
Total objects: 2
```

静态函数成员

如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。静态成员函数即使在类对象不存在的情况下也能被调用，静态函数只要使用类名加范围解析运算符 **::** 就可以访问。

静态成员函数只能访问静态数据成员，不能访问其他静态成员函数和类外部的其他函数。

静态成员函数有一个类范围，他们不能访问类的 this 指针。您可以使用静态成员函数来判断类的某些对象是否已被创建。

下面的实例有助于更好地理解静态函数成员的概念：

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // 每次创建对象时增加 1
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // 长度
      double breadth;    // 宽度
      double height;     // 高度
};

// 初始化类 Box 的静态成员
int Box::objectCount = 0;

int main(void)
{

   // 在创建对象之前输出对象的总数
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // 声明 box1
   Box Box2(8.5, 6.0, 2.0);    // 声明 box2

   // 在创建对象之后输出对象的总数
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

C++ 继承

面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了重用代码功能和提高执行时间的效果。

当创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为基类，新建的类称为派生类。

继承代表了 **is a** 关系。例如，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。

基类 & 派生类

一个类可以派生自多个类，这意味着，它可以从多个基类继承数据和函数。定义一个派生类，我们使用一个类派生列表来指定基类。类派生列表以一个或多个基类命名，形式如下：

```
class derived-class: access-specifier base-class
```

其中，访问修饰符 access-specifier 是 **public**、**protected** 或 **private** 其中的一个，base-class 是之前定义过的某个类的名称。如果未使用访问修饰符 access-specifier，则默认为 private。

假设有一个基类 **Shape**，**Rectangle** 是它的派生类，如下所示：

```
#include <iostream>

using namespace std;

// 基类
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};

// 派生类
class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // 输出对象的面积
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total area:  35
```

访问控制和继承

派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 private。

我们可以根据访问权限总结出不同的访问类型，如下所示：

访问	public	protected	private
同一个类	yes	yes	yes
派生类	yes	yes	no
外部的类	yes	no	no

一个派生类继承了所有的基类方法，但下列情况除外：

- 基类的构造函数、析构函数和拷贝构造函数。
- 基类的重载运算符。
- 基类的友元函数。

继承类型

当一个类派生自基类，该基类可以被继承为 **public**、**protected** 或 **private** 几种类型。继承类型是通过上面讲解的访问修饰符 access-specifier 来指定的。

我们几乎不使用 **protected** 或 **private** 继承，通常使用 **public** 继承。当使用不同类型的继承时，遵循以下几个规则：

- 公有继承（**public**）：当一个类派生自公有基类时，基类的公有成员也是派生类的公有成员，基类的保护成员也是派生类的保护成员，基类的私有成员不能直接被派生类访问，但是可以通过调用基类的公有和保护成员来访问。
- 保护继承（**protected**）：当一个类派生自保护基类时，基类的公有和保护成员将成为派生类的保护成员。
- 私有继承（**private**）：当一个类派生自私有基类时，基类的公有和保护成员将成为派生类的私有成员。

多重继承

C++ 类可以从多个类继承成员，语法如下：

```
class derived-class: access baseA, access baseB....
```

其中，访问修饰符 access 是 **public**、**protected** 或 **private** 其中的一个，用来修饰每个基类，各个基类之间用逗号分隔，如上所示。现在让我们一起看看下面的实例：

```
#include <iostream>

using namespace std;

// 基类 Shape
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};

// 基类 PaintCost
class PaintCost
{
   public:
      int getCost(int area)
      {
         return area * 70;
      }
};

// 派生类
class Rectangle: public Shape, public PaintCost
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // 输出对象的面积
   cout << "Total area: " << Rect.getArea() << endl;

   // 输出总花费
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total area:  35
Total paint cost: $2450
```

C++ 重载运算符和重载函数

C++ 允许在同一作用域中的某个函数和运算符指定多个定义，分别称为函数重载和运算符重载。

重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

当您调用一个重载函数或重载运算符时，编译器通过把您所使用的参数类型与定义中的参数类型进行比较，决定选用最合适的定义。选择最合适的重载函数或重载运算符的过程，称为重载决策。

C++ 中的函数重载

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。您不能仅通过返回类型的不同来重载函数。

下面的实例中，同名函数 **print()** 被用于输出不同的数据类型：

```
#include <iostream>
using namespace std;

class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }

      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void)
{
   printData pd;

   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

C++ 中的运算符重载

您可以重定义或重载大部分 C++ 内置的运算符。这样，您就能使用自定义类型的运算符。

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

```
Box  operator+(const  Box&);
```

声明加法运算符用于把两个 Box 对象相加，返回最终的 Box 对象。大多数的重载运算符可被定义为普通的非成员函数或这被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

```
Box  operator+(const  Box&,  const  Box&);
```

下面的实例使用成员函数演示了运算符重载的概念。在这里，对象作为参数进行传递，对象的属性使用 **this** 运算符进行访问，如下所示：

```
#include <iostream>
using namespace std;

class Box
{
   public:

      double getVolume(void)
      {
         return length * breadth * height;
      }
      void setLength( double len )
      {
          length = len;
      }

      void setBreadth( double bre )
      {
          breadth = bre;
      }

      void setHeight( double hei )
      {
          height = hei;
      }
      // 重载 + 运算符，用于把两个 Box 对象相加
      Box operator+(const Box& b)
      {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
```

```
      }
   private:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度
};
// 程序的主函数
int main( )
{
   Box Box1;                // 声明 Box1，类型为 Box
   Box Box2;                // 声明 Box2，类型为 Box
   Box Box3;                // 声明 Box3，类型为 Box
   double volume = 0.0;     // 把体积存储在该变量中

   // Box1 详述
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // Box2 详述
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // Box1 的体积
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // Box2 的体积
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // 把两个对象相加，得到 Box3
   Box3 = Box1 + Box2;

   // Box3 的体积
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

可重载运算符/不可重载运算符

下面是可重载的运算符列表：

<table>
<tr><th>+</th><th>-</th><th>*</th><th>/</th><th>%</th><th>^</th></tr>
<tr><td>&</td><td>|</td><td>~</td><td>!</td><td>,</td><td>=</td></tr>
<tr><td><</td><td>></td><td><=</td><td>>=</td><td>++</td><td>--</td></tr>
<tr><td><<</td><td>>></td><td>==</td><td>!=</td><td>&&</td><td>||</td></tr>
<tr><td>+=</td><td>-=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td></tr>
<tr><td>|=</td><td>*=</td><td><<=</td><td>>>=</td><td>[]</td><td>()</td></tr>
<tr><td>-></td><td>->*</td><td>new</td><td>new []</td><td>delete</td><td>delete []</td></tr>
</table>

下面是不可重载的运算符列表：

::	.*	.	?:

运算符重载实例

下面提供了各种运算符重载的实例，帮助您更好地理解重载的概念。

序号	运算符和实例
1	一元运算符重载
2	二元运算符重载
3	关系运算符重载
4	输入/输出运算符重载
5	++ 和 -- 运算符重载
6	赋值运算符重载
7	函数调用运算符 () 重载
8	下标运算符 [] 重载
9	类成员访问运算符 -> 重载

C++ 一元运算符重载

一元运算符只对一个操作数进行操作，下面是一元运算符的实例：

- 递增运算符（ ++ ）和递减运算符（ -- ）
- 一元减运算符，即负号（ - ）
- 逻辑非运算符（ ! ）

一元运算符通常出现在它们所操作的对象的左边，比如 !obj、-obj 和 ++obj，但有时它们也可以作为后缀，比如 obj++ 或 obj--。

下面的实例演示了如何重载一元减运算符（ - ）。

```
#include <iostream>
using namespace std;

class Distance
{
   private:
      int feet;             // 0 到无穷
      int inches;           // 0 到 12
   public:
      // 所需的构造函数
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // 显示距离的方法
      void displayDistance()
      {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // 重载负运算符（ - ）
      Distance operator- ()
      {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};
int main()
{
   Distance D1(11, 10), D2(-5, 11);

   -D1;                     // 取相反数
   D1.displayDistance();    // 距离 D1

   -D2;                     // 取相反数
   D2.displayDistance();    // 距离 D2

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
F:  -11 I:-10
F:  5 I:-11
```

希望上面的实例能够帮您更好地理解一元运算符重载的概念，类似地，您可以尝试重载逻辑非运算符（！）。

C++ 二元运算符重载

二元运算符需要两个参数，下面是二元运算符的实例。我们平常使用的加运算符（+）、减运算符（-）、乘运算符（*）和除运算符（/）都属于二元运算符。就像加(+)运算符。

下面的实例演示了如何重载加运算符（+）。类似地，您也可以尝试重载减运算符（-）和除运算符（/）。

```
#include <iostream>
using namespace std;

class Box
{
   double length;      // 长度
   double breadth;     // 宽度
   double height;      // 高度
public:

   double getVolume(void)
   {
      return length * breadth * height;
   }
   void setLength( double len )
   {
       length = len;
   }

   void setBreadth( double bre )
   {
       breadth = bre;
   }

   void setHeight( double hei )
   {
       height = hei;
   }
   // 重载 + 运算符，用于把两个 Box 对象相加
   Box operator+(const Box& b)
   {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};
// 程序的主函数
int main( )
{
   Box Box1;                // 声明 Box1，类型为 Box
   Box Box2;                // 声明 Box2，类型为 Box
   Box Box3;                // 声明 Box3，类型为 Box
   double volume = 0.0;     // 把体积存储在该变量中

   // Box1 详述
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // Box2 详述
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);
```

```
    // Box1 的体积
    volume = Box1.getVolume();
    cout << "Volume of Box1 : " << volume <<endl;

    // Box2 的体积
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume <<endl;

    // 把两个对象相加，得到 Box3
    Box3 = Box1 + Box2;

    // Box3 的体积
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume <<endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

C++ 关系运算符重载

C++ 语言支持各种关系运算符（ <、 >、 <=、 >=、 == 等等），它们可用于比较 C++ 内置的数据类型。

您可以重载任何一个关系运算符，重载后的关系运算符可用于比较类的对象。

下面的实例演示了如何重载 < 运算符，类似地，您也可以尝试重载其他的关系运算符。

```
#include <iostream>
using namespace std;

class Distance
{
   private:
      int feet;             // 0 到无穷
      int inches;           // 0 到 12
   public:
      // 所需的构造函数
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // 显示距离的方法
      void displayDistance()
      {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // 重载负运算符（ - ）
      Distance operator- ()
      {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
      // 重载小于运算符（ < ）
      bool operator <(const Distance& d)
      {
         if(feet < d.feet)
         {
            return true;
         }
         if(feet == d.feet && inches < d.inches)
         {
            return true;
         }
         return false;
      }
};
int main()
{
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 )
   {
      cout << "D1 is less than D2 " << endl;
   }
   else
   {
      cout << "D2 is less than D1 " << endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
D2 is less than D1
```

C++ 输入/输出运算符重载

C++ 能够使用流提取运算符 >> 和流插入运算符 << 来输入和输出内置的数据类型。您可以重载流提取运算符和流插入运算符来操作对象等用户自定义的数据类型。

在这里，有一点很重要，我们需要把运算符重载函数声明为类的友元函数，这样我们就能不用创建对象而直接调用函数。

下面的实例演示了如何重载提取运算符 >> 和插入运算符 <<。

```
#include <iostream>
using namespace std;

class Distance
{
   private:
      int feet;             // 0 到无穷
      int inches;           // 0 到 12
   public:
      // 所需的构造函数
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      friend ostream &operator<<( ostream &output,
                                  const Distance &D )
      {
         output << "F : " << D.feet << " I : " << D.inches;
         return output;
      }

      friend istream &operator>>( istream  &input, Distance &D )
      {
         input >> D.feet >> D.inches;
         return input;
      }
};
int main()
{
   Distance D1(11, 10), D2(5, 11), D3;

   cout << "Enter the value of object : " << endl;
   cin >> D3;
   cout << "First Distance : " << D1 << endl;
   cout << "Second Distance :" << D2 << endl;
   cout << "Third Distance :" << D3 << endl;


   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
$./a.out
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```

C++ ++ 和 -- 运算符重载

递增运算符（ ++ ）和递减运算符（ -- ）是 C++ 语言中两个重要的一元运算符。

下面的实例演示了如何重载递增运算符（ ++ ），包括前缀和后缀两种用法。类似地，您也可以尝试重载递减运算符（ -- ）。

```
t hours;             // 0 到 23
      int minutes;           // 0 到 59
   public:
      // 所需的构造函数
      Time(){
         hours = 0;
         minutes = 0;
      }
      Time(int h, int m){
         hours = h;
         minutes = m;
      }
      // 显示时间的方法
      void displayTime()
      {
         cout << "H: " << hours << " M:" << minutes <<endl;
      }
      // 重载前缀递增运算符（ ++ ）
      Time operator++ ()
      {
         ++minutes;          // 对象加 1
         if(minutes >= 60)
         {
            ++hours;
            minutes -= 60;
         }
         return Time(hours, minutes);
      }
      // 重载后缀递增运算符（ ++ ）
      Time operator++( int )
      {
         // 保存原始值
         Time T(hours, minutes);
         // 对象加 1
         ++minutes;
         if(minutes >= 60)
         {
            ++hours;
            minutes -= 60;
         }
         // 返回旧的原始值
         return T;
      }
};
int main()
{
   Time T1(11, 59), T2(10,40);

   ++T1;                    // T1 加 1
   T1.displayTime();        // 显示 T1
   ++T1;                    // T1 再加 1
   T1.displayTime();        // 显示 T1

   T2++;                    // T2 加 1
   T2.displayTime();        // 显示 T2
   T2++;                    // T2 再加 1
   T2.displayTime();        // 显示 T2
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

C++ 赋值运算符重载

就像其他运算符一样，您可以重载赋值运算符（ = ），用于创建一个对象，比如拷贝构造函数。

下面的实例演示了如何重载赋值运算符。

```
#include <iostream>
using namespace std;

class Distance
{
   private:
      int feet;             // 0 到无穷
      int inches;           // 0 到 12
   public:
      // 所需的构造函数
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      void operator=(const Distance &D )
      {
         feet = D.feet;
         inches = D.inches;
      }
      // 显示距离的方法
      void displayDistance()
      {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }

};
int main()
{
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // 使用赋值运算符
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

C++ 函数调用运算符 () 重载

函数调用运算符 () 可以被重载用于类的对象。当重载 () 时，您不是创造了一种新的调用函数的方式，相反地，这是创建一个可以传递任意数目参数的运算符函数。

下面的实例演示了如何重载函数调用运算符 ()。

```
/ 所需的构造函数
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // 重载函数调用运算符
      Distance operator()(int a, int b, int c)
      {
         Distance D;
         // 进行随机计算
         D.feet = a + c + 10;
         D.inches = b + c + 100 ;
         return D;
      }
      // 显示距离的方法
      void displayDistance()
      {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }

};
int main()
{
   Distance D1(11, 10), D2;

   cout << "First Distance : ";
   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :";
   D2.displayDistance();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

C++ 下标运算符 [] 重载

下标操作符 [] 通常用于访问数组元素。这个运算符符可被重载用于增强操作 C++ 数组的功能。

下面的实例演示了如何重载下标运算符 []。

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay
{
   private:
      int arr[SIZE];
   public:
      safearay()
      {
         register int i;
         for(i = 0; i < SIZE; i++)
         {
           arr[i] = i;
         }
      }
      int &operator[](int i)
      {
          if( i > SIZE )
          {
              cout << "Index out of bounds" <<endl;
              // 返回第一个元素
              return arr[0];
          }
          return arr[i];
      }
};
int main()
{
   safearay A;

   cout << "Value of A[2] : " << A[2] <<endl;
   cout << "Value of A[5] : " << A[5]<<endl;
   cout << "Value of A[12] : " << A[12]<<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

C++ 类成员访问运算符 -> 重载

类成员访问运算符（ -> ）可以被重载，但它较为麻烦。它被定义用于为一个类赋予"指针"行为。运算符 -> 必须是一个成员函数。如果使用了 -> 运算符，返回类型必须是指针或者是类的对象。

运算符 -> 通常与指针引用运算符 * 结合使用，用于实现"智能指针"的功能。这些指针是行为与正常指针相似的对象，唯一不同的是，当您通过指针访问对象时，它们会执行其他的任务。比如，当指针销毁时，或者当指针指向另一个对象时，会自动删除对象。

间接引用运算符 -> 可被定义为一个一元后缀运算符。也就是说，给出一个类：

```
class Ptr{
   //...
   X * operator->();
};
```

类 **Ptr** 的对象可用于访问类 **X** 的成员，使用方式与指针的用法十分相似。例如：

```
void f(Ptr p )
{
   p->m = 10 ; // (p.operator->())->m = 10
}
```

语句 p->m 被解释为 (p.operator->())->m。同样地，下面的实例演示了如何重载类成员访问运算符 ->。

```
#include <iostream>
#include <vector>
using namespace std;

// 假设一个实际的类
class Obj {
   static int i, j;
public:
   void f() const { cout << i++ << endl; }
   void g() const { cout << j++ << endl; }
};

// 静态成员定义
int Obj::i = 10;
int Obj::j = 12;

// 为上面的类实现一个容器
class ObjContainer {
   vector<Obj*> a;
public:
   void add(Obj* obj)
   {
      a.push_back(obj);  // 调用向量的标准方法
   }
   friend class SmartPointer;
};
```

```
// 实现智能指针，用于访问类 Obj 的成员
class SmartPointer {
   ObjContainer oc;
   int index;
public:
   SmartPointer(ObjContainer& objc)
   {
       oc = objc;
       index = 0;
   }
   // 返回值表示列表结束
   bool operator++() // 前缀版本
   {
     if(index >= oc.a.size()) return false;
     if(oc.a[++index] == 0) return false;
     return true;
   }
   bool operator++(int) // 后缀版本
   {
      return operator++();
   }
   // 重载运算符 ->
   Obj* operator->() const
   {
     if(!oc.a[index])
     {
        cout << "Zero value";
        return (Obj*)0;
     }
     return oc.a[index];
   }
};

int main() {
   const int sz = 10;
   Obj o[sz];
   ObjContainer oc;
   for(int i = 0; i < sz; i++)
   {
       oc.add(&o[i]);
   }
   SmartPointer sp(oc); // 创建一个迭代器
   do {
      sp->f(); // 智能指针调用
      sp->g();
   } while(sp++);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
10
12
11
13
12
14
13
15
14
16
15
17
16
18
17
19
18
20
19
21
```

C++ 多态

多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。

C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

下面的实例中，基类 Shape 被派生为两个类，如下所示：

```
#include <iostream>
using namespace std;

class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// 程序的主函数
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // 存储矩形的地址
   shape = &rec;
   // 调用矩形的求面积函数 area
   shape->area();

   // 存储三角形的地址
   shape = &tri;
   // 调用三角形的求面积函数 area
   shape->area();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Parent class area
Parent class area
```

导致错误输出的原因是，调用函数 area() 被编译器设置为基类中的版本，这就是所谓的静态多态，或静态链接 - 函数调用在程序执行前就准备好了。有时候这也被称为早绑定，因为 area() 函数在程序编译期间就已经设置好了。

但现在，让我们对程序稍作修改，在 Shape 类中，area() 的声明前放置关键字 **virtual**，如下所示：

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

修改后，当编译和执行前面的实例代码时，它会产生以下结果：

```
Rectangle class area
Triangle class area
```

此时，编译器看的是指针的内容，而不是它的类型。因此，由于 tri 和 rec 类的对象的地址存储在 *shape 中，所以会调用各自的 area() 函数。

正如您所看到的，每个子类都有一个函数 area() 的独立实现。这就是多态的一般使用方式。有了多态，您可以有多个不同的类，都带有同一个名称但具有不同实现的函数，函数的参数甚至可以是相同的。

虚函数

虚函数 是在基类中使用关键字 **virtual** 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。

我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为动态链接，或后期绑定。

纯虚函数

您可能想要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是您在基类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。

我们可以把基类中的虚函数 area() 改写如下：

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      // pure virtual function
      virtual int area() = 0;
};
```

= 0 告诉编译器，函数没有主体，上面的虚函数是纯虚函数。

C++ 数据抽象

数据抽象是指，只向外界提供关键信息，并隐藏其后台的实现细节，即只表现必要的信息而不呈现细节。

数据抽象是一种依赖于接口和实现分离的编程（设计）技术。

让我们举一个现实生活中的真实例子，比如一台电视机，您可以打开和关闭、切换频道、调整音量、添加外部组件（如喇叭、录像机、DVD 播放器），但是您不知道它的内部实现细节，也就是说，您并不知道它是如何通过缆线接收信号，如何转换信号，并最终显示在屏幕上。

因此，我们可以说电视把它的内部实现和外部接口分离开了，您无需知道它的内部实现原理，直接通过它的外部接口（比如电源按钮、遥控器、声量控制器）就可以操控电视。

现在，让我们言归正传，就 C++ 编程而言，C++ 类为数据抽象提供了可能。它们向外界提供了大量用于操作对象数据的公共方法，也就是说，外界实际上并不清楚类的内部实现。

例如，您的程序可以调用 **sort()** 函数，而不需要知道函数中排序数据所用到的算法。实际上，函数排序的底层实现会因库的版本不同而有所差异，只要接口不变，函数调用就可以照常工作。

在 C++ 中，我们使用类来定义我们自己的抽象数据类型（ADT）。您可以使用类 **ostream** 的 **cout** 对象来输出数据到标准输出，如下所示：

```
#include <iostream>
using namespace std;

int main( )
{
   cout << "Hello C++" <<endl;
   return 0;
}
```

在这里，您不需要理解 **cout** 是如何在用户的屏幕上显示文本。您只需要知道公共接口即可，cout 的底层实现可以自由改变。

访问标签强制抽象

在 C++ 中，我们使用访问标签来定义类的抽象接口。一个类可以包含零个或多个访问标签：

- 使用公共标签定义的成员都可以访问该程序的所有部分。一个类型的数据抽象视图是由它的公共成员来定义的。
- 使用私有标签定义的成员无法访问到使用类的代码。私有部分对使用类型的代码隐藏了实现细节。

访问标签出现的频率没有限制。每个访问标签指定了紧随其后的成员定义的访问级别。指定的访问级别会一直有效，直到遇到下一个访问标签或者遇到类主体的关闭右括号为止。

数据抽象的好处

数据抽象有两个重要的优势：

- 类的内部受到保护，不会因无意的用户级错误导致对象状态受损。
- 类实现可能随着时间的推移而发生变化，以便应对不断变化的需求，或者应对那些要求不改变用户级代码的错误报告。

如果只在类的私有部分定义数据成员，编写该类的作者就可以随意更改数据。如果实现发生改变，则只需要检查类的代码，看看这个改变会导致哪些影响。如果数据是公有的，则任何直接访问旧表示形式的数据成员的函数都可能受到影响。

数据抽象的实例

C++ 程序中，任何带有公有和私有成员的类都可以作为数据抽象的实例。请看下面的实例：

```
#include <iostream>
using namespace std;

class Adder{
   public:
      // 构造函数
      Adder(int i = 0)
      {
        total = i;
      }
      // 对外的接口
      void addNum(int number)
      {
          total += number;
      }
      // 对外的接口
      int getTotal()
      {
          return total;
      };
   private:
      // 对外隐藏的数据
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total 60
```

上面的类把数字相加，并返回总和。公有成员 **addNum** 和 **getTotal** 是对外的接口，用户需要知道它们以便使用类。私有成员 **total** 是用户不需要了解的，但又是类能正常工作所必需的。

设计策略

抽象把代码分离为接口和实现。所以在设计组件时，必须保持接口独立于实现，这样，如果改变底层实现，接口也将保持不变。

在这种情况下，不管任何程序使用接口，接口都不会受到影响，只需要将最新的实现重新编译即可。

C++ 数据封装

所有的 C++ 程序都有以下两个基本要素：

- 程序语句（代码）：这是程序中执行动作的部分，它们被称为函数。
- 程序数据：数据是程序的信息，会受到程序函数的影响。

封装是面向对象编程中的把数据和操作数据的函数绑定在一起的一个概念，这样能避免受到外界的干扰和误用，从而确保了安全。数据封装引申出了另一个重要的 OOP 概念，即数据隐藏。

数据封装是一种把数据和操作数据的函数捆绑在一起的机制，数据抽象是一种仅向用户暴露接口而把具体的实现细节隐藏起来的机制。

C++ 通过创建类来支持封装和数据因此。我们已经知道，类包含私有成员（private）、保护成员（protected）和公有成员（public）成员。默认情况下，在类中定义的所有项目都是私有的。例如：

```
class Box
{
   public:
      double getVolume(void)
      {
         return length * breadth * height;
      }
   private:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度
};
```

变量 length、breadth 和 height 都是私有的（private）。这意味着它们只能被 Box 类中的其他成员访问，而不能被程序中其他部分访问。这是实现封装的一种方式。

为了使类中的成员变成公有的（即，程序中的其他部分也能访问），必须在这些成员前使用 **public** 关键字进行声明。所有定义在 public 标识符后边的变量或函数可以被程序中所有其他的函数访问。

把一个类定义为另一个类的友元类，会暴露实现细节，从而降低了封装性。理想的做法是尽可能地对外隐藏每个类的实现细节。

数据封装的实例

C++ 程序中，任何带有公有和私有成员的类都可以作为数据封装和数据抽象的实例。请看下面的实例：

```
#include <iostream>
using namespace std;

class Adder{
   public:
      // 构造函数
      Adder(int i = 0)
      {
        total = i;
      }
      // 对外的接口
      void addNum(int number)
      {
          total += number;
      }
      // 对外的接口
      int getTotal()
      {
          return total;
      };
   private:
      // 对外隐藏的数据
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total 60
```

上面的类把数字相加，并返回总和。公有成员 **addNum** 和 **getTotal** 是对外的接口，用户需要知道它们以便使用类。私有成员 **total** 是对外隐藏的，用户不需要了解它，但它又是类能正常工作所必需的。

设计策略

通常情况下，我们都会设置类成员状态为私有（private），除非我们真的需要将其暴露，这样才能保证良好的封装性。

这通常应用于数据成员，但它同样适用于所有成员，包括虚函数。

C++ 接口（抽象类）

接口描述了类的行为和功能，而不需要完成类的特定实现。

C++ 接口是使用抽象类来实现的，抽象类与数据抽象互不混淆，数据抽象是一个把实现细节与相关的数据分离开的概念。

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 "= 0" 来指定的，如下所示：

```
class Box
{
   public:
      // 纯虚函数
      virtual double getVolume() = 0;
   private:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度
};
```

设计抽象类（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为接口使用。如果试图实例化一个抽象类的对象，会导致编译错误。

因此，如果一个 ABC 的子类需要被实例化，则必须实现每个虚函数，这也意味着 C++ 支持使用 ABC 声明接口。如果没有在派生类中重载纯虚函数，就尝试实例化该类的对象，会导致编译错误。

可用于实例化对象的类被称为具体类。

抽象类的实例

请看下面的实例，基类 Shape 提供了一个接口 **getArea()**，在两个派生类 Rectangle 和 Triangle 中分别实现了 **getArea()**：

```
#include <iostream>

using namespace std;

// 基类
class Shape
{
public:
   // 提供接口框架的纯虚函数
   virtual int getArea() = 0;
   void setWidth(int w)
   {
      width = w;
   }
   void setHeight(int h)
   {
      height = h;
   }
protected:
   int width;
   int height;
};

// 派生类
class Rectangle: public Shape
{
public:
   int getArea()
   {
      return (width * height);
   }
};
class Triangle: public Shape
{
public:
   int getArea()
   {
      return (width * height)/2;
   }
};

int main(void)
{
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);
   // 输出对象的面积
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);
   // 输出对象的面积
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total  Rectangle area:  35
Total  Triangle area:  17
```

从上面的实例中，我们可以看到一个抽象类是如何定义一个接口 getArea()，两个派生类是如何通过不同的计算面积的算法来实现这个相同的函数。

设计策略

面向对象的系统可能会使用一个抽象基类为所有的外部应用程序提供一个适当的、通用的、标准化的接口。然后，派生类通过继承抽象基类，就把所有类似的操作都继承下来。

外部应用程序提供的功能（即公有函数）在抽象基类中是以纯虚函数的形式存在的。这些纯虚函数在相应的派生类中被实现。

这个架构也使得新的应用程序可以很容易地被添加到系统中，即使是在系统被定义之后依然可以如此。

C++ 高级

C++ 文件和流

到目前为止，我们已经使用了 **iostream** 标准库，它提供了 **cin** 和 **cout** 方法分别用于从标准输入读取流和向标准输出写入流。

本教程介绍如何从文件读取流和向文件写入流。这就需要用到 C++ 中另一个标准库 **fstream**，它定义了三个新的数据类型：

数据类型	描述
ofstream	该数据类型表示输出文件流，用于创建文件并向文件写入信息。
ifstream	该数据类型表示输入文件流，用于从文件读取信息。
fstream	该数据类型通常表示文件流，且同时具有 ofstream 和 ifstream 两种功能，这意味着它可以创建文件，向文件写入信息，从文件读取信息。

要在 C++ 中进行文件处理，必须在 C++ 源代码文件中包含头文件 <iostream> 和 <fstream>。

打开文件

在从文件读取信息或者向文件写入信息之前，必须先打开文件。**ofstream** 和 **fstream** 对象都可以用来打开文件进行写操作，如果只需要打开文件进行读操作，则使用 **ifstream** 对象。

下面是 open() 函数的标准语法，open() 函数是 fstream、ifstream 和 ofstream 对象的一个成员。

```
void open(const  char  *filename, ios::openmode mode);
```

在这里，**open()** 成员函数的第一参数指定要打开的文件的名称和位置，第二个参数定义文件被打开的模式。

模式标志	描述
ios::app	追加模式。所有写入都追加到文件末尾。
ios::ate	文件打开后定位到文件末尾。
ios::in	打开文件用于读取。
ios::out	打开文件用于写入。
ios::trunc	如果该文件已经存在，其内容将在打开文件之前被截断，即把文件长度设为 0。

您可以把以上两种或两种以上的模式结合使用。例如，如果您想要以写入模式打开文件，并希望截断文件，以防文件已存在，那么您可以使用下面的语法：

```
ofstream outfile;
outfile.open("file.dat", ios::out  | ios::trunc );
```

类似地，您如果想要打开一个文件用于读写，可以使用下面的语法：

```
fstream  afile;
afile.open("file.dat", ios::out  | ios::in  );
```

关闭文件

当 C++ 程序终止时，它会自动关闭刷新所有流，释放所有分配的内存，并关闭所有打开的文件。但程序员应该养成一个好习惯，在程序终止前关闭所有打开的文件。

下面是 close() 函数的标准语法，close() 函数是 fstream、ifstream 和 ofstream 对象的一个成员。

```
void close();
```

写入文件

在 C++ 编程中，我们使用流插入运算符（ << ）向文件写入信息，就像使用该运算符输出信息到屏幕上一样。唯一不同的是，在这里您使用的是 **ofstream** 或 **fstream** 对象，而不是 **cout** 对象。

读取文件

在 C++ 编程中，我们使用流提取运算符（ >> ）从文件读取信息，就像使用该运算符从键盘输入信息一样。唯一不同的是，在这里您使用的是 **ifstream** 或 **fstream** 对象，而不是 **cin** 对象。

读取 & 写入实例

下面的 C++ 程序以读写模式打开一个文件。在向文件 afile.dat 写入用户输入的信息之后，程序从文件读取信息，并将其输出到屏幕上：

```
#include <fstream>
#include <iostream>
using namespace std;

int main ()
{

   char data[100];

   // 以写模式打开文件
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // 向文件写入用户输入的数据
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // 再次向文件写入用户输入的数据
   outfile << data << endl;

   // 关闭打开的文件
   outfile.close();

   // 以读模式打开文件
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // 在屏幕上写入数据
   cout << data << endl;

   // 再次从文件读取数据，并显示它
   infile >> data;
   cout << data << endl;

   // 关闭打开的文件
   infile.close();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列输入和输出：

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

上面的实例中使用了 cin 对象的附加函数，比如 getline()函数从外部读取一行，ignore() 函数会忽略掉之前读语句留下的多余字符。

文件位置指针

istream 和 **ostream** 都提供了用于重新定位文件位置指针的成员函数。这些成员函数包括关于 istream 的 **seekg**（"seek get"）和关于 ostream 的 **seekp**（"seek put"）。

seekg 和 seekp 的参数通常是一个长整型。第二个参数可以用于指定查找方向。查找方向可以是 **ios::beg**（默认的，从流的开头开始定位），也可以是 **ios::cur**（从流的当前位置开始定位），也可以是 **ios::end**（从流的末尾开始定位）。

文件位置指针是一个整数值，指定了从文件的起始位置到指针所在位置的字节数。下面是关于定位 "get" 文件位置指针的实例：

```
// 定位到 fileObject 的第 n 个字节（假设是 ios::beg）
fileObject.seekg( n );

// 把文件的读指针从 fileObject 当前位置向后移 n 个字节
fileObject.seekg( n, ios::cur );

// 把文件的读指针从 fileObject 末尾往回移 n 个字节
fileObject.seekg( n, ios::end );

// 定位到 fileObject 的末尾
fileObject.seekg( 0, ios::end );
```

C++ 异常处理

异常是程序在执行期间产生的问题。C++ 异常是指在程序运行时发生的特殊情况，比如尝试除以零的操作。

异常提供了一种转移程序控制权的方式。C++ 异常处理涉及到三个关键字：**try**、**catch**、**throw**。

- **throw:** 当问题出现时，程序会抛出一个异常。这是通过使用 **throw** 关键字来完成的。
- **catch:** 在您想要处理问题的地方，通过异常处理程序捕获异常。**catch** 关键字用于捕获异常。
- **try:** **try** 块中的代码标识将被激活的特定异常。它后面通常跟着一个或多个 catch 块。

如果有一个块抛出一个异常，捕获异常的方法会使用 **try** 和 **catch** 关键字。try 块中放置可能抛出异常的代码，try 块中的代码被称为保护代码。使用 try/catch 语句的语法如下所示：

```
try
{
   // 保护代码
}catch( ExceptionName e1 )
{
   // catch 块
}catch( ExceptionName e2 )
{
   // catch 块
}catch( ExceptionName eN )
{
   // catch 块
}
```

如果 **try** 块在不同的情境下会抛出不同的异常，这个时候可以尝试罗列多个 **catch** 语句，用于捕获不同类型的异常。

抛出异常

您可以使用 **throw** 语句在代码块中的任何地方抛出异常。throw 语句的操作数可以是任意的表达式，表达式的结果的类型决定了抛出的异常的类型。

以下是尝试除以零时抛出异常的实例：

```
double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

捕获异常

catch 块跟在 **try** 块后面，用于捕获异常。您可以指定想要捕捉的异常类型，这是由 catch 关键字后的括号内的异常声明决定的。

```
try
{
   // 保护代码
}catch( ExceptionName e )
{
  // 处理 ExceptionName 异常的代码
}
```

上面的代码会捕获一个类型为 **ExceptionName** 的异常。如果您想让 catch 块能够处理 try 块抛出的任何类型的异常，则必须在异常声明的括号内使用省略号 ...，如下所示：

```
try
{
   // 保护代码
}catch(...)
{
  // 能处理任何异常的代码
}
```

下面是一个实例，抛出一个除以零的异常，并在 catch 块中捕获该异常。

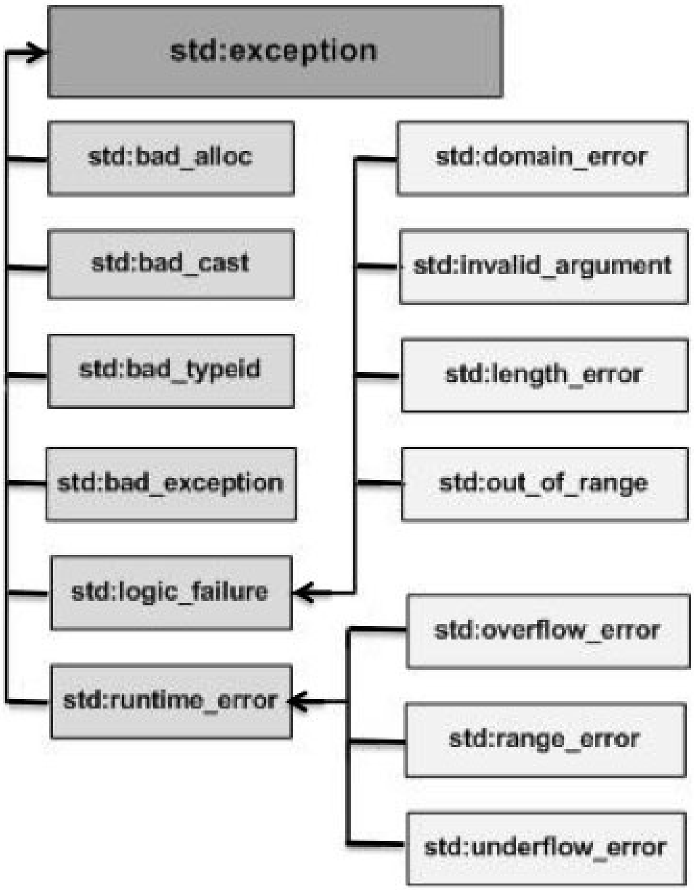
```
#include <iostream>
using namespace std;

double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;

   try {
     z = division(x, y);
     cout << z << endl;
   }catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

由于我们抛出了一个类型为 **const char*** 的异常，因此，当捕获该异常时，我们必须在 catch 块中使用 const char*。当上面的代码被编译和执行时，它会产生下列结果：

```
Division by zero condition!
```

C++ 标准的异常

C++ 提供了一系列标准的异常，定义在 **<exception>** 中，我们可以在程序中使用这些标准的异常。它们是以父子类层次结构组织起来的，如下所示：



下表是对上面层次结构中出现的每个异常的说明：

异常	描述
std::exception	该异常是所有标准 C++ 异常的父类。
std::bad_alloc	该异常可以通过 **new** 抛出。
std::bad_cast	该异常可以通过 **dynamic_cast** 抛出。
std::bad_exception	这在处理 C++ 程序中无法预期的异常时非常有用。
std::bad_typeid	该异常可以通过 **typeid** 抛出。
std::logic_error	理论上可以通过读取代码来检测到的异常。
std::domain_error	当使用了一个无效的数学域时，会抛出该异常。
std::invalid_argument	当使用了无效的参数时，会抛出该异常。
std::length_error	当创建了太长的 std::string 时，会抛出该异常。
std::out_of_range	该异常可以通过方法抛出，例如 std::vector 和 std::bitset<>::operator。
std::runtime_error	理论上不可以通过读取代码来检测到的异常。
std::overflow_error	当发生数学上溢时，会抛出该异常。
std::range_error	当尝试存储超出范围的值时，会抛出该异常。
std::underflow_error	当发生数学下溢时，会抛出该异常。

定义新的异常

您可以通过继承和重载 **exception** 类来定义新的异常。下面的实例演示了如何使用 std::exception 类来实现自己的异常：

```
#include <iostream>
#include <exception>
using namespace std;

struct MyException : public exception
{
  const char * what () const throw ()
  {
    return "C++ Exception";
  }
};

int main()
{
  try
  {
    throw MyException();
  }
  catch(MyException& e)
  {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
    //其他的错误
  }
}
```

这将产生以下结果：

```
MyException caught
C++ Exception
```

在这里，**what()** 是异常类提供的一个公共方法，它已被所有子异常类重载。这将返回异常产生的原因。

C++ 动态内存

了解动态内存在 C++ 中是如何工作的是成为一名合格的 C++ 程序员必不可少的。C++ 程序中的内存分为两个部分：

- 栈：在函数内部声明的所有变量都将占用栈内存。
- 堆：这是程序中未使用的内存，在程序运行时可用于动态分配内存。

很多时候，您无法提前预知需要多少内存来存储某个定义变量中的特定信息，所需内存的大小需要在运行时才能确定。

在 C++ 中，您可以使用特殊的运算符为给定类型的变量在运行时分配堆内的内存，这会返回所分配的空间地址。这种运算符即 **new** 运算符。

如果您不需要动态分配内存，可以使用 **delete** 运算符，删除之前由 new 运算符分配的内存。

new 和 delete 运算符

下面是使用 new 运算符来为任意的数据类型动态分配内存的通用语法：

```
new data-type;
```

在这里，**data-type** 可以是包括数组在内的任意内置的数据类型，也可以是包括类或结构在内的用户自定义的任何数据类型。让我们先来看下内置的数据类型。例如，我们可以定义一个指向 double 类型的指针，然后请求内存，该内存在执行时被分配。我们可以按照下面的语句使用 **new** 运算符来完成这点：

```
double* pvalue  = NULL; // 初始化为 null 的指针
pvalue  = new double;   // 为变量请求内存
```

如果自由存储区已被用完，可能无法成功分配内存。所以建议检查 new 运算符是否返回 NULL 指针，并采取以下适当的操作：

```
double* pvalue  = NULL;
if( !(pvalue  = new double ))
{
   cout << "Error: out of memory." <<endl;
   exit(1);

}
```

malloc() 函数在 C 语言中就出现了，在 C++ 中仍然存在，但建议尽量不要使用 malloc() 函数。new 与 malloc() 函数相比，其主要的优点是，new 不只是分配了内存，它还创建了对象。

在任何时候，当您觉得某个已经动态分配内存的变量不再需要使用时，您可以使用 delete 操作符释放它所占用的内存，如下所示：

```
delete pvalue;        // 释放 pvalue 所指向的内存
```

下面的实例中使用了上面的概念，演示了如何使用 new 和 delete 运算符：

```
#include <iostream>
using namespace std;

int main ()
{
   double* pvalue  = NULL; // 初始化为 null 的指针
   pvalue  = new double;   // 为变量请求内存

   *pvalue = 29494.99;     // 在分配的地址存储值
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // 释放内存

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of pvalue : 29495
```

数组的动态内存分配

假设我们要为一个字符数组（一个有 20 个字符的字符串）分配内存，我们可以使用上面实例中的语法来为数组动态地分配内存，如下所示：

```
char* pvalue  = NULL;   // 初始化为 null 的指针
pvalue  = new char[20]; // 为变量请求内存
```

要删除我们刚才创建的数组，语句如下：

```
delete [] pvalue;        // 删除 pvalue 所指向的数组
```

下面是 new 操作符的通用语法，可以为多维数组分配内存，如下所示：

```
double** pvalue  = NULL;     // 初始化为 null 的指针
pvalue  = new double [3][4]; // 为一个 3x4 数组分配内存
```

释放多维数组内存的语法与二维数组一样：

```
delete [] pvalue;        // 删除 pvalue 所指向的数组
```

对象的动态内存分配

对象与简单的数据类型没有什么不同。例如，请看下面的代码，我们将使用一个对象数组来理清这一概念：

```
#include <iostream>
using namespace std;

class Box
{
   public:
      Box() { 
         cout << "调用构造函数！" <<endl; 
      }
      ~Box() { 
         cout << "调用析构函数！" <<endl; 
      }
};

int main( )
{
   Box* myBoxArray = new Box[4];

   delete [] myBoxArray; // Delete array

   return 0;
}
```

如果要为一个包含四个 Box 对象的数组分配内存，构造函数将被调用 4 次，同样地，当删除这些对象时，析构函数也将被调用相同的次数（4次）。

当上面的代码被编译和执行时，它会产生下列结果：

```
调用构造函数！
调用构造函数！
调用构造函数！
调用构造函数！
调用析构函数！
调用析构函数！
调用析构函数！
调用析构函数！
```

C++ 命名空间

假设这样一种情况，当一个班上有两个名叫 Zara 的学生时，为了明确区分它们，我们在使用名字之外，不得不使用一些额外的信息，比如他们的家庭住址，或者他们父母的名字等等。

同样的情况也出现在 C++ 应用程序中。例如，您可能会写一个名为 xyz() 的函数，在另一个可用的库中也存在一个相同的函数 xyz()。这样，编译器就无法判断您所使用的是哪一个 xyz() 函数。

因此，引入了命名空间这个概念，专门用于解决上面的问题，它可作为附加信息来区分不同库中相同名称的函数、类、变量等。使用了命名空间即定义了上下文。本质上，命名空间就是定义了一个范围。

定义命名空间

命名空间的定义使用关键字 **namespace**，后跟命名空间的名称，如下所示：

```
namespace namespace_name {
   // 代码声明
}
```

为了调用带有命名空间的函数或变量，需要在前面加上命名空间的名称，如下所示：

```
name::code;  // code 可以是变量或函数
```

让我们来看看命名空间如何为变量或函数等实体定义范围：

```
#include <iostream>
using namespace std;

// 第一个命名空间
namespace first_space{
   void func(){
      cout << "Inside first_space" << endl;
   }
}
// 第二个命名空间
namespace second_space{
   void func(){
      cout << "Inside second_space" << endl;
   }
}
int main ()
{

   // 调用第一个命名空间中的函数
   first_space::func();

   // 调用第二个命名空间中的函数
   second_space::func();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

using 指令

您可以使用 **using namespace** 指令，这样在使用命名空间时就可以不用在前面加上命名空间的名称。这个指令会告诉编译器，后续的代码将使用指定的命名空间中的名称。

```
#include <iostream>
using namespace std;

// 第一个命名空间
namespace first_space{
   void func(){
      cout << "Inside first_space" << endl;
   }
}
// 第二个命名空间
namespace second_space{
   void func(){
      cout << "Inside second_space" << endl;
   }
}
using namespace first_space;
int main ()
{

   // 调用第一个命名空间中的函数
   func();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
```

using 指令也可以用来指定命名空间中的特定项目。例如，如果您只打算使用 std 命名空间中的 cout 部分，您可以使用如下的语句：

```
using std::cout;
```

随后的代码中，在使用 cout 时就可以不用加上命名空间名称作为前缀，但是 **std** 命名空间中的其他项目仍然需要加上命名空间名称作为前缀，如下所示：

```
#include <iostream>
using std::cout;

int main ()
{

   cout << "std::endl is used with std!" << std::endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
std::endl is used with std!
```

using 指令引入的名称遵循正常的范围规则。名称从使用 **using** 指令开始是可见的，直到该范围结束。此时，在范围以外定义的同名实体是隐藏的。

不连续的命名空间

命名空间可以定义在几个不同的部分中，因此命名空间是由几个单独定义的部分组成的。一个命名空间的各个组成部分可以分散在多个文件中。

所以，如果命名空间中的某个组成部分需要请求定义在另一个文件中的名称，则仍然需要声明该名称。下面的命名空间定义可以是定义一个新的命名空间，也可以是为已有的命名空间增加新的元素：

```
namespace namespace_name {
   // 代码声明
}
```

嵌套的命名空间

命名空间可以嵌套，您可以在一个命名空间中定义另一个命名空间，如下所示：

```
namespace namespace_name1 {
   // 代码声明
   namespace namespace_name2 {
      // 代码声明
   }
}
```

您可以通过使用 :: 运算符来访问嵌套的命名空间中的成员：

```
// 访问 namespace_name2 中的成员
using namespace namespace_name1::namespace_name2;

// 访问 namespace:name1 中的成员
using namespace namespace_name1;
```

在上面的语句中，如果使用的是 namespace_name1，那么在该范围内 namespace_name2 中的元素也是可用的，如下所示：

```
#include <iostream>
using namespace std;

// 第一个命名空间
namespace first_space{
   void func(){
      cout << "Inside first_space" << endl;
   }
   // 第二个命名空间
   namespace second_space{
      void func(){
         cout << "Inside second_space" << endl;
      }
   }
}
using namespace first_space::second_space;
int main ()
{

   // 调用第二个命名空间中的函数
   func();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside second_space
```

C++ 模板

模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的方式编写代码。

模板是创建泛型类或函数的蓝图或公式。库容器，比如迭代器和算法，都是泛型编程的例子，它们都使用了模板的概念。

每个容器都有一个单一的定义，比如 向量，我们可以定义许多不同类型的向量，比如 **vector <int>** 或 **vector <string>**。

您可以使用模板来定义函数和类，接下来让我们一起来看看如何使用。

函数模板

模板函数定义的一般形式如下所示：

```
template <class type> ret-type func-name(parameter list)
{
   // 函数的主体
}
```

在这里，type 是函数所使用的数据类型的占位符名称。这个名称可以在函数定义中使用。

下面是函数模板的实例，返回两个数种的最大值：

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
    return a < b ? b:a;
}
int main ()
{

    int i = 39;
    int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;

    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;

    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

类模板

正如我们定义函数模板一样，我们也可以定义类模板。泛型类声明的一般形式如下所示：

```
template <class type> class class-name {
.
.
.
}
```

在这里，**type** 是占位符类型名称，可以在类被实例化的时候进行指定。您可以使用一个逗号分隔的列表来定义多个泛型数据类型。

下面的实例定义了类 Stack<>，并实现了泛型方法来对元素进行入栈出栈操作：

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
  private:
    vector<T> elems;     // 元素

  public:
    void push(T const&);  // 入栈
    void pop();               // 出栈
    T top() const;            // 返回栈顶元素
    bool empty() const{       // 如果为空则返回真。
        return elems.empty();
    }
};

template <class T>
void Stack<T>::push (T const& elem)
{
    // 追加传入元素的副本
    elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::pop(): empty stack");
    }
    // 删除最后一个元素
    elems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::top(): empty stack");
    }
    // 返回最后一个元素的副本
    return elems.back();
}

int main()
{
    try {
        Stack<int>         intStack;  // int 类型的栈
        Stack<string> stringStack;    // string 类型的栈

        // 操作 int 类型的栈
        intStack.push(7);
        cout << intStack.top() <<endl;

        // 操作 string 类型的栈
        stringStack.push("hello");
        cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (exception const& ex) {
        cerr << "Exception: " << ex.what() <<endl;
        return -1;
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
7
hello
Exception: Stack<>::pop(): empty stack
```

C++ 预处理器

预处理器是一些指令，指示编译器在实际编译之前所需完成的预处理。

所有的预处理器指令都是以井号（#）开头，只有空格字符可以出现在预处理指令之前。预处理指令不是 C++ 语句，所以它们不会以分号（;）结尾。

我们已经看到，之前所有的实例中都有 **#include** 指令。这个宏用于把头文件包含到源文件中。

C++ 还支持很多预处理指令，比如 #include、#define、#if、#else、#line 等，让我们一起看看这些重要指令。

#define 预处理

#define 预处理指令用于创建符号常量。该符号常量通常称为宏，指令的一般形式是：

```
#define macro-name replacement-text
```

当这一行代码出现在一个文件中时，在该文件中后续出现的所有宏都将会在程序编译之前被替换为 replacement-text。例如：

```
sing namespace std;

#define PI 3.14159

int main ()
{

    cout << "Value of PI :" << PI << endl;

    return 0;
}
```

现在，让我们测试这段代码，看看预处理的结果。假设源代码文件已经存在，接下来使用 -E 选项进行编译，并把结果重定向到 test.p。现在，如果您查看 test.p 文件，将会看到它已经包含大量的信息，而且在文件底部的值被改为如下：

```
$gcc -E test.cpp > test.p

...
int main ()
{

    cout << "Value of PI :" << 3.14159 << endl;

    return 0;
}
```

函数宏

您可以使用 #define 来定义一个带有参数的宏，如下所示：

```
#include <iostream>
using namespace std;

#define MIN(a,b) (((a)<(b)) ? a : b)

int main ()
{
   int i, j;
   i = 100;
   j = 30;
   cout <<"The minimum is " << MIN(i, j) << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The minimum is  30
```

条件编译

有几个指令可以用来有选择地对部分程序源代码进行编译。这个过程被称为条件编译。

条件预处理器的结构与 if 选择结构很像。请看下面这段预处理器的代码：

```
#ifndef NULL
   #define NULL 0
#endif
```

您可以只在调试时进行编译，调试开关可以使用一个宏来实现，如下所示：

```
#ifdef DEBUG
   cerr <<"Variable x = " << x << endl;
#endif
```

如果在指令 #ifdef DEBUG 之前已经定义了符号常量 DEBUG，则会对程序中的 **cerr** 语句进行编译。您可以使用 #if 0 语句注释掉程序的一部分，如下所示：

```
#if 0
   不进行编译的代码
#endif
```

让我们尝试下面的实例：

```
#include <iostream>
using namespace std;
#define DEBUG

#define MIN(a,b) (((a)<(b)) ? a : b)

int main ()
{
   int i, j;
   i = 100;
   j = 30;
#ifdef DEBUG
   cerr <<"Trace: Inside main function" << endl;
#endif

#if 0
   /* 这是注释部分 */
   cout << MKSTR(HELLO C++) << endl;
#endif

   cout <<"The minimum is " << MIN(i, j) << endl;

#ifdef DEBUG
   cerr <<"Trace: Coming out of main function" << endl;
#endif
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Trace: Inside main function
The minimum is 30
Trace: Coming out of main function
```

和 ## 运算符

和 ## 预处理运算符在 C++ 和 ANSI/ISO C 中都是可用的。# 运算符会把 replacement-text 令牌转换为用引号引起来的字符串。

请看下面的宏定义：

```
#include <iostream>
using namespace std;

#define MKSTR( x ) #x

int main ()
{
    cout << MKSTR(HELLO C++) << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
HELLO C++
```

让我们来看看它是如何工作的。不难理解，C++ 预处理器把下面这行：

```
cout << MKSTR(HELLO C++)  << endl;
```

转换成了：

```
cout <<  "HELLO C++"  << endl;
```

运算符用于连接两个令牌。下面是一个实例：

```
#define CONCAT( x, y ) x ## y
```

当 CONCAT 出现在程序中时，它的参数会被连接起来，并用来取代宏。例如，程序中 CONCAT(HELLO, C++) 会被替换为 "HELLO C++"，如下面实例所示。

```
#include <iostream>
using namespace std;

#define concat(a, b) a ## b
int main()
{
   int xy = 100;

   cout << concat(x, y);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
100
```

让我们来看看它是如何工作的。不难理解，C++ 预处理器把下面这行：

```
cout << concat(x, y);
```

转换成了：

```
cout << xy;
```

C++ 中的预定义宏

C++ 提供了下表所示的一些预定义宏：

宏	描述
__LINE__	这会在程序编译时包含当前行号。
__FILE__	这会在程序编译时包含当前文件名。
__DATE__	这会包含一个形式为 month/day/year 的字符串，它表示把源文件转换为目标代码的日期。
__TIME__	这会包含一个形式为 hour:minute:second 的字符串，它表示程序被编译的时间。

让我们看看上述这些宏的实例：

```
#include <iostream>
using namespace std;

int main ()
{
    cout << "Value of __LINE__ : " << __LINE__ << endl;
    cout << "Value of __FILE__ : " << __FILE__ << endl;
    cout << "Value of __DATE__ : " << __DATE__ << endl;
    cout << "Value of __TIME__ : " << __TIME__ << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of __LINE__ : 6
Value of __FILE__ : test.cpp
Value of __DATE__ : Feb 28 2011
Value of __TIME__ : 18:52:48
```

C++ 信号处理

信号是由操作系统传给进程的中断，会提早终止一个程序。在 UNIX、LINUX、Mac OS X 或 Windows 系统上，可以通过按 Ctrl+C 产生中断。

有些信号不能被程序捕获，但是下表所列信号可以在程序中捕获，并可以基于信号采取适当的动作。这些信号是定义在 C++ 头文件 <csignal> 中。

信号	描述
SIGABRT	程序的异常终止，如调用 **abort**。
SIGFPE	错误的算术运算，比如除以零或导致溢出的操作。
SIGILL	检测非法指令。
SIGINT	接收到交互注意信号。
SIGSEGV	非法访问内存。
SIGTERM	发送到程序的终止请求。

signal() 函数

C++ 信号处理库提供了 **signal** 函数，用来捕获突发事件。以下是 signal() 函数的语法：

```
void (*signal (int sig, void (*func)(int)))(int);
```

这个函数接收两个参数：第一个参数是一个整数，代表了信号的编号；第二个参数是一个指向信号处理函数的指针。

让我们编写一个简单的 C++ 程序，使用 signal() 函数捕获 SIGINT 信号。不管您想在程序中捕获什么信号，您都必须使用 **signal** 函数来注册信号，并将其与信号处理程序相关联。看看下面的实例：

```
#include <iostream>
#include <csignal>

using namespace std;

void signalHandler( int signum )
{
    cout << "Interrupt signal (" << signum << ") received.\n";

    // 清理并关闭
    // 终止程序

   exit(signum);

}

int main ()
{
    // 注册信号 SIGINT 和信号处理程序
    signal(SIGINT, signalHandler);

    while(1){
       cout << "Going to sleep...." << endl;
       sleep(1);
    }

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Going to sleep....
Going to sleep....
Going to sleep....
```

现在，按 Ctrl+C 来中断程序，您会看到程序捕获信号，程序打印如下内容并退出：

```
Going to sleep....
Going to sleep....
Going to sleep....
Interrupt signal (2) received.
```

raise() 函数

您可以使用函数 **raise()** 生成信号，该函数带有一个整数信号编号作为参数，语法如下：

```
int raise (signal sig);
```

在这里，**sig** 是要发送的信号的编号，这些信号包括：SIGINT、SIGABRT、SIGFPE、SIGILL、SIGSEGV、SIGTERM、SIGHUP。以下是我们使用 raise() 函数内部生成信号的实例：

```
#include <iostream>
#include <csignal>

using namespace std;

void signalHandler( int signum )
{
    cout << "Interrupt signal (" << signum << ") received.\n";

    // 清理并关闭
    // 终止程序

   exit(signum);

}

int main ()
{
    int i = 0;
    // 注册信号 SIGINT 和信号处理程序
    signal(SIGINT, signalHandler);

    while(++i){
       cout << "Going to sleep...." << endl;
       if( i == 3 ){
          raise( SIGINT);
       }
       sleep(1);
    }

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果，并会自动退出：

```
Going to sleep....
Going to sleep....
Going to sleep....
Interrupt signal (2) received.
```

C++ 多线程

多线程是多任务处理的一种特殊形式，多任务处理允许让电脑同时运行两个或两个以上的程序。在一般情况下，有两种类型的多任务处理：基于进程和基于线程。

基于进程的多任务处理处理的是程序的并发执行。基于线程的多任务处理的是同一程序的片段的并发执行。

多线程程序包含可以同时运行的两个或多个部分。这样的程序中的每个部分称为一个线程，每个线程定义了一个单独的执行路径。

C++ 不包含多线程应用程序的任何内置支持。相反，它完全依赖于操作系统来提供此功能。

本教程假设您使用的是 Linux 操作系统，我们要使用 POSIX 编写多线程 C++ 程序。POSIX Threads 或 Pthreads 提供的 API 可在多种类 Unix POSIX 系统上可用，比如 FreeBSD、NetBSD、GNU/Linux、Mac OS X 和 Solaris。

创建线程

有下面的例程，我们可以用它来创建一个 POSIX 线程：

```
#include <iostream>
using namespace std;

int main ()
{
    cout << "Value of __LINE__ : " << __LINE__ << endl;
    cout << "Value of __FILE__ : " << __FILE__ << endl;
    cout << "Value of __DATE__ : " << __DATE__ << endl;
    cout << "Value of __TIME__ : " << __TIME__ << endl;

    return 0;
}
```

在这里，**pthread_create** 创建一个新的线程，并让它可执行。这个例程可在代码内的任何地方被调用任意次数。下面是关于参数的说明：

参数	描述
thread	一个不透明的、唯一的标识符，用来标识例程返回的新线程。
attr	一个不透明的属性对象，可以被用来设置线程属性。您可以指定线程属性对象，也可以使用默认值 NULL。
start_routine	C++ 例程，一旦线程被创建就会执行。
arg	一个可能传递给 start_routine 的参数。它必须通过把引用作为指针强制转换为 void 类型进行传递。如果没有传递参数，则使用 NULL。

一个进程可以创建的最大线程数是依赖于实现的。线程一旦被创建，就是同等的，而且可以创建其他线程。线程之间没有隐含层次或依赖。

终止线程

有下面的例程，我们可以用它来终止一个 POSIX 线程：

```
#include <pthread.h>
pthread_exit (status)
```

在这里，**pthread_exit** 用于显式地退出一个线程。通常情况下，pthread_exit() 例程是在线程完成工作后无需继续存在时被调用。

如果 main() 是在它所创建的线程之前结束，并通过 pthread_exit() 退出，那么其他线程将继续执行。否则，它们将在 main() 结束时自动被终止。

实例

这个简单的实例代码使用 pthread_create() 例程创建了 5 个线程。每个线程打印一个 "Hello World!" 消息，然后调用 pthread_exit() 终止线程。

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   cout << "Hello World! Thread ID, " << tid << endl;
   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;
   for( i=0; i < NUM_THREADS; i++ ){
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL,
                          PrintHello, (void *)i);
      if (rc){
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

使用 -lpthread 库编译下面的程序：

```
$gcc test.cpp -lpthread
```

现在，执行程序，将产生下列结果：

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

向线程传递参数

这个实例演示了如何通过结构传递多个参数。您可以在线程回调中传递任意的数据类型，因为它指向 void，如下面的实例所示：

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS     5

struct thread_data{
   int  thread_id;
   char *message;
};

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;

   my_data = (struct thread_data *) threadarg;

   cout << "Thread ID : " << my_data->thread_id ;
   cout << " Message : " << my_data->message << endl;

   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NUM_THREADS];
   struct thread_data td[NUM_THREADS];
   int rc;
   int i;

   for( i=0; i < NUM_THREADS; i++ ){
      cout <<"main() : creating thread, " << i << endl;
      td[i].thread_id = i;
      td[i].message = "This is message";
      rc = pthread_create(&threads[i], NULL,
                          PrintHello, (void *)&td[i]);
      if (rc){
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 2 Message : This is message
Thread ID : 0 Message : This is message
Thread ID : 1 Message : This is message
Thread ID : 4 Message : This is message
```

连接和分离线程

有以下两个例程，我们可以用它们来连接或分离线程：

```
pthread_join (threadid, status)
pthread_detach (threadid)
```

pthread_join() 子例程阻碍调用例程，直到指定的 threadid 线程终止为止。当创建一个线程时，它的某个属性会定义它是否是可连接的（joinable）或可分离的（detached）。只有创建时定义为可连接的线程才可以被连接。如果线程创建时被定义为可分离的，则它永远也不能被连接。

这个实例演示了如何使用 pthread_join() 例程来等待线程的完成。

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>

using namespace std;

#define NUM_THREADS     5

void *wait(void *t)
{
   int i;
   long tid;

   tid = (long)t;

   sleep(1);
   cout << "Sleeping in thread " << endl;
   cout << "Thread with id : " << tid << "  ...exiting " << endl;
   pthread_exit(NULL);
}

int main ()
{
   int rc;
   int i;
   pthread_t threads[NUM_THREADS];
   pthread_attr_t attr;
   void *status;

   // 初始化并设置线程为可连接的（joinable）
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for( i=0; i < NUM_THREADS; i++ ){
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL, wait, (void *)i );
      if (rc){
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }

   // 删除属性，并等待其他线程
   pthread_attr_destroy(&attr);
   for( i=0; i < NUM_THREADS; i++ ){
      rc = pthread_join(threads[i], &status);
      if (rc){
         cout << "Error:unable to join," << rc << endl;
         exit(-1);
      }
      cout << "Main: completed thread id :" << i ;
      cout << "  exiting with status :" << status << endl;
   }

   cout << "Main: program exiting." << endl;
   pthread_exit(NULL);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 .... exiting
Sleeping in thread
Thread with id : 1 .... exiting
Sleeping in thread
Thread with id : 2 .... exiting
Sleeping in thread
Thread with id : 3 .... exiting
Sleeping in thread
Thread with id : 4 .... exiting
Main: completed thread id :0  exiting with status :0
Main: completed thread id :1  exiting with status :0
Main: completed thread id :2  exiting with status :0
Main: completed thread id :3  exiting with status :0
Main: completed thread id :4  exiting with status :0
Main: program exiting.
```

C++ Web 编程

什么是 CGI？

- 公共网关接口（CGI），是一套标准，定义了信息是如何在 Web 服务器和客户端脚本之间进行交换的。
- CGI 规范目前是由 NCSA 维护的，NCSA 定义 CGI 如下：
- 公共网关接口（CGI），是一种用于外部网关程序与信息服务器（如 HTTP 服务器）对接的接口标准。
- 目前的版本是 CGI/1.1，CGI/1.2 版本正在推进中。

Web 浏览

为了更好地了解 CGI 的概念，让我们点击一个超链接，浏览一个特定的网页或 URL，看看会发生什么。

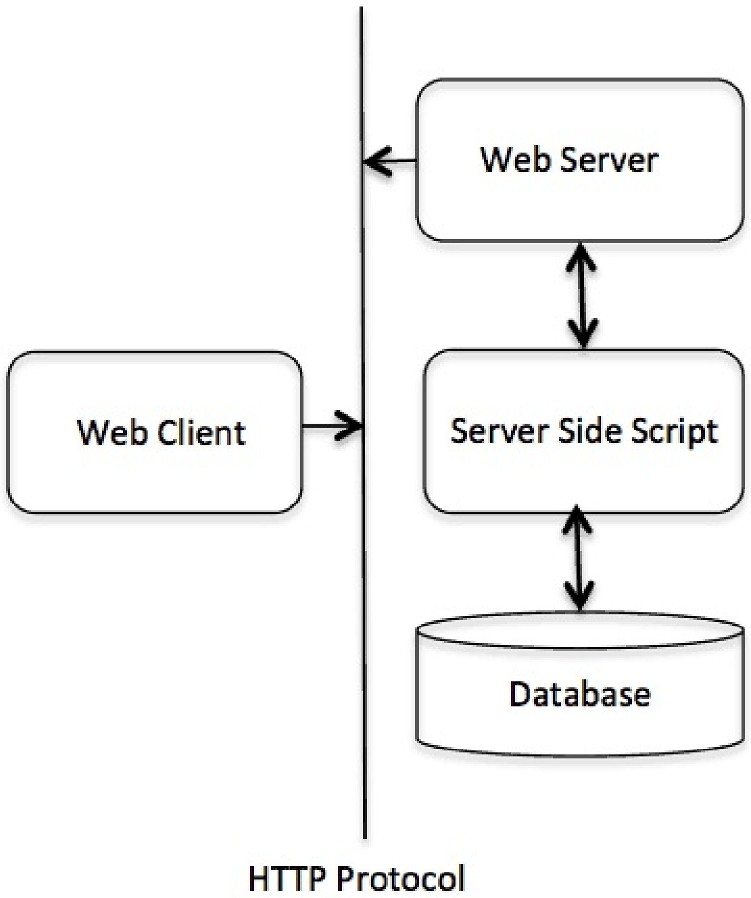
- 您的浏览器联系上 HTTP Web 服务器，并请求 URL，即文件名。
- Web 服务器将解析 URL，并查找文件名。如果找到请求的文件，Web 服务器会把文件发送回浏览器，否则发送一条错误消息，表明您请求了一个错误的文件。
- Web 浏览器从 Web 服务器获取响应，并根据接收到的响应来显示文件或错误消息。

然而，以这种方式搭建起来的 HTTP 服务器，不管何时请求目录中的某个文件，HTTP 服务器发送回来的不是该文件，而是以程序形式执行，并把执行产生的输出发送回浏览器显示出来。

公共网关接口（CGI），是使得应用程序（称为 CGI 程序或 CGI 脚本）能够与 Web 服务器以及客户端进行交互的标准协议。这些 CGI 程序可以用 Python、PERL、Shell、C 或 C++ 等进行编写。

CGI 架构图

下图演示了 CGI 的架构：



Web 服务器配置

在您进行 CGI 编程之前，请确保您的 Web 服务器支持 CGI，并已配置成可以处理 CGI 程序。所有由 HTTP 服务器执行的 CGI 程序，都必须在预配置的目录中。该目录称为 CGI 目录，按照惯例命名为 /var/www/cgi-bin。虽然 CGI 文件是 C++ 可执行文件，但是按照惯例它的扩展名是 **.cgi**。

默认情况下，Apache Web 服务器会配置在 /var/www/cgi-bin 中运行 CGI 程序。如果您想指定其他目录来运行 CGI 脚本，您可以在 httpd.conf 文件中修改以下部分：

```
<Directory "/var/www/cgi-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>

<Directory "/var/www/cgi-bin">
Options All
</Directory>
```

在这里，我们假设已经配置好 Web 服务器并能成功运行，你可以运行任意的 CGI 程序，比如 Perl 或 Shell 等。

第一个 CGI 程序

请看下面的 C++ 程序：

```
#include <iostream>
using namespace std;

int main ()
{

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Hello World - 第一个 CGI 程序</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<h2>Hello World! 这是我的第一个 CGI 程序</h2>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

编译上面的代码，把可执行文件命名为 cplusplus.cgi，并把这个文件保存在 /var/www/cgi-bin 目录中。在运行 CGI 程序之前，请使用 **chmod 755 cplusplus.cgi** UNIX 命令来修改文件模式，确保文件可执行。访问可执行文件，您会看到下面的输出：

Hello World! 这是我的第一个 CGI 程序

上面的 C++ 程序是一个简单的程序，把它的输出写在 STDOUT 文件上，即显示在屏幕上。在这里，值得注意一点，第一行输出 **Content-type:text/html\r\n\r\n**。这一行发送回浏览器，并指定要显示在浏览器窗口上的内容类型。您必须理解 CGI 的基本概念，这样才能进一步使用 Python 编写更多复杂的 CGI 程序。C++ CGI 程序可以与任何其他外部的系统（如 RDBMS）进行交互。

HTTP 头信息

行 **Content-type:text/html\r\n\r\n** 是 HTTP 头信息的组成部分，它被发送到浏览器，以便更好地理解页面内容。HTTP 头信息的形式如下：

```
HTTP 字段名称: 字段内容

例如
Content-type: text/html\r\n\r\n
```

还有一些其他的重要的 HTTP 头信息，这些在您的 CGI 编程中都会经常被用到。

头信息	描述
Content-type:	MIME 字符串，定义返回的文件格式。例如 Content-type:text/html。
Expires: Date	信息变成无效的日期。浏览器使用它来判断一个页面何时需要刷新。一个有效的日期字符串的格式应为 01 Jan 1998 12:00:00 GMT。
Location: URL	这个 URL 是指应该返回的 URL，而不是请求的 URL。你可以使用它来重定向一个请求到任意的文件。
Last-modified: Date	资源的最后修改日期。
Content-length: N	要返回的数据的长度，以字节为单位。浏览器使用这个值来表示一个文件的预计下载时间。
Set-Cookie: String	通过 *string* 设置 cookie。

CGI 环境变量

所有的 CGI 程序都可以访问下列的环境变量。这些变量在编写 CGI 程序时扮演了非常重要的角色。

变量名	描述
CONTENT_TYPE	内容的数据类型。当客户端向服务器发送附加内容时使用。例如，文件上传等功能。
CONTENT_LENGTH	查询的信息长度。只对 POST 请求可用。
HTTP_COOKIE	以键 & 值对的形式返回设置的 cookies。
HTTP_USER_AGENT	用户代理请求标头字段，递交用户发起请求的有关信息，包含了浏览器的名称、版本和其他平台性的附加信息。
PATH_INFO	CGI 脚本的路径。
QUERY_STRING	通过 GET 方法发送请求时的 URL 编码信息，包含 URL 中问号后面的参数。
REMOTE_ADDR	发出请求的远程主机的 IP 地址。这在日志记录和认证时是非常有用的。
REMOTE_HOST	发出请求的主机的完全限定名称。如果此信息不可用，则可以用 REMOTE_ADDR 来获取 IP 地址。
REQUEST_METHOD	用于发出请求的方法。最常见的方法是 GET 和 POST。
SCRIPT_FILENAME	CGI 脚本的完整路径。
SCRIPT_NAME	CGI 脚本的名称。
SERVER_NAME	服务器的主机名或 IP 地址。
SERVER_SOFTWARE	服务器上运行的软件的名称和版本。

下面的 CGI 程序列出了所有的 CGI 变量。

```
#include <iostream>
#include <stdlib.h>
using namespace std;

const string ENV[ 24 ] = {
        "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
        "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
        "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
        "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
        "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
        "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
        "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
        "SERVER_NAME","SERVER_PORT","SERVER_PROTOCOL",
        "SERVER_SIGNATURE","SERVER_SOFTWARE" };

int main ()
{

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>CGI 环境变量</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<table border = \"0\" cellspacing = \"2\">";

   for ( int i = 0; i < 24; i++ )
   {
       cout << "<tr><td>" << ENV[ i ] << "</td><td>";
       // 尝试检索环境变量的值
       char *value = getenv( ENV[ i ].c_str() );
       if ( value != 0 ){
         cout << value;
       }else{
         cout << "环境变量不存在。";
       }
       cout << "</td></tr>\n";
   }
   cout << "</table><\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

C++ CGI 库

在真实的实例中，您需要通过 CGI 程序执行许多操作。这里有一个专为 C++ 程序而编写的 CGI 库，我们可以从 ftp://ftp.gnu.org/gnu/cgicc/ 上下载这个 CGI 库，并按照下面的步骤安装库：

```
$tar xzf cgicc-X.X.X.tar.gz
$cd cgicc-X.X.X/ $./configure --prefix=/usr
$make
$make install
```

您可以点击 C++ CGI Lib Documentation，查看相关的库文档。

GET 和 POST 方法

您可能有遇到过这样的情况，当您需要从浏览器传递一些信息到 Web 服务器，最后再传到 CGI 程序。通常浏览器会使用两种方法把这个信息传到 Web 服务器，分别是 GET 和 POST 方法。

使用 GET 方法传递信息

GET 方法发送已编码的用户信息追加到页面请求中。页面和已编码信息通过 ? 字符分隔开，如下所示：

```
http://www.test.com/cgi-bin/cpp.cgi?key1=value1&key2=value2
```

GET 方法是默认的从浏览器向 Web 服务器传信息的方法，它会在浏览器的地址栏中生成一串很长的字符串。当您向服务器传密码或其他一些敏感信息时，不要使用 GET 方法。GET 方法有大小限制，在一个请求字符串中最多可以传 1024 个字符。

当使用 GET 方法时，是使用 QUERY_STRING http 头来传递信息，在 CGI 程序中可使用 QUERY_STRING 环境变量来访问。

您可以通过在 URL 后跟上简单连接的键值对，也可以通过使用 HTML <FORM> 标签的 GET 方法来传信息。

简单的 URL 实例：Get 方法

下面是一个简单的 URL，使用 GET 方法传递两个值给 hello_get.py 程序。

/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI

下面的实例生成 **cpp_get.cgi** CGI 程序，用于处理 Web 浏览器给出的输入。通过使用 C++ CGI 库，可以很容易地访问传递的信息：

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>使用 GET 和 POST 方法</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("first_name");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "名：" << **fi << endl;
   }else{
      cout << "No text entered for first name" << endl;
   }
   cout << "<br/>\n";
   fi = formData.getElement("last_name");
   if( !fi->isEmpty() &&fi != (*formData).end()) {
      cout << "姓：" << **fi << endl;
   }else{
      cout << "No text entered for last name" << endl;
   }
   cout << "<br/>\n";

   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

现在，编译上面的程序，如下所示：

```
$g++  -o cpp_get.cgi cpp_get.cpp -lcgicc
```

生成 cpp_get.cgi，并把它放在 CGI 目录中，并尝试使用下面的链接进行访问：

```
/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI
```

这会产生以下结果：

```
名：ZARA
姓：ALI
```

简单的表单实例：GET 方法

下面是一个简单的实例，使用 HTML 表单和提交按钮传递两个值。我们将使用相同的 CGI 脚本 cpp_get.cgi 来处理输入。

```
<form action="/cgi-bin/cpp_get.cgi" method="get">
名：<input type="text" name="first_name">  <br />

姓：<input type="text" name="last_name" />
<input type="submit" value="提交" />
</form>
```

下面是上述表单的实际输出，请输入名和姓，然后点击提交按钮查看结果。

使用 POST 方法传递信息

一个更可靠的向 CGI 程序传递信息的方法是 POST 方法。这种方法打包信息的方式与 GET 方法相同，不同的是，它不是把信息以文本字符串形式放在 URL 中的？之后进行传递，而是把它以单独的消息形式进行传递。该消息是以标准输入的形式传给 CGI 脚本的。

我们同样使用 cpp_get.cgi 程序来处理 POST 方法。让我们以同样的例子，通过使用 HTML 表单和提交按钮来传递两个值，只不过这次我们使用的不是 GET 方法，而是 POST 方法，如下所示：

```
<form action="/cgi-bin/cpp_checkbox.cgi"
         method="POST"
         target="_blank">
<input type="checkbox" name="maths" value="on" /> 数学
<input type="checkbox" name="physics" value="on" /> 物理
<input type="submit" value="选择学科" />
</form>
```

向 CGI 程序传递复选框数据

当需要选择多个选项时，我们使用复选框。

下面的 HTML 代码实例是一个带有两个复选框的表单：

```
<form action="/cgi-bin/cpp_checkbox.cgi"
         method="POST"
         target="_blank">
<input type="checkbox" name="maths" value="on" /> 数学
<input type="checkbox" name="physics" value="on" /> 物理
<input type="submit" value="选择学科" />
</form>
```

下面的 C++ 程序会生成 cpp_checkbox.cgi 脚本，用于处理 Web 浏览器通过复选框给出的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;
   bool maths_flag, physics_flag;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>向 CGI 程序传递复选框数据</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   maths_flag = formData.queryCheckbox("maths");
   if( maths_flag ) {
      cout << "Maths Flag: ON " << endl;
   }else{
      cout << "Maths Flag: OFF " << endl;
   }
   cout << "<br/>\n";

   physics_flag = formData.queryCheckbox("physics");
   if( physics_flag ) {
      cout << "Physics Flag: ON " << endl;
   }else{
      cout << "Physics Flag: OFF " << endl;
   }
   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

向 CGI 程序传递单选按钮数据

当只需要选择一个选项时，我们使用单选按钮。

下面的 HTML 代码实例是一个带有两个单选按钮的表单：

```
<form action="/cgi-bin/cpp_radiobutton.cgi"
         method="post"
         target="_blank">
<input type="radio" name="subject" value="maths"
                                    checked="checked"/> 数学
<input type="radio" name="subject" value="physics" /> 物理
<input type="submit" value="选择学科" />
</form>
```

下面的 C++ 程序会生成 cpp_radiobutton.cgi 脚本，用于处理 Web 浏览器通过单选按钮给出的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>向 CGI 程序传递单选按钮数据</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("subject");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Radio box selected: " << **fi << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

向 CGI 程序传递文本区域数据

当需要向 CGI 程序传递多行文本时，我们使用 TEXTAREA 元素。

下面的 HTML 代码实例是一个带有 TEXTAREA 框的表单：

```
<form action="/cgi-bin/cpp_textarea.cgi"
         method="post"
         target="_blank">
<textarea name="textcontent" cols="40" rows="4">
请在这里输入文本...
</textarea>
<input type="submit" value="提交" />
</form>
```

下面的 C++ 程序会生成 cpp_textarea.cgi 脚本，用于处理 Web 浏览器通过文本区域给出的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>向 CGI 程序传递文本区域数据</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("textcontent");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Text Content: " << **fi << endl;
   }else{
      cout << "No text entered" << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

向 CGI 程序传递下拉框数据

当有多个选项可用，但只能选择一个或两个选项时，我们使用下拉框。

下面的 HTML 代码实例是一个带有下拉框的表单：

```
<form action="/cgi-bin/cpp_dropdown.cgi"
                       method="post" target="_blank">
<select name="dropdown">
<option value="Maths" selected>数学</option>
<option value="Physics">物理</option>
</select>
<input type="submit" value="提交"/>
</form>
```

下面的 C++ 程序会生成 cpp_dropdown.cgi 脚本，用于处理 Web 浏览器通过下拉框给出的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>向 CGI 程序传递下拉框数据</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("dropdown");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Value Selected: " << **fi << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

在 CGI 中使用 Cookies

HTTP 协议是一种无状态的协议。但对于一个商业网站，它需要在不同页面间保持会话信息。例如，一个用户在完成多个页面的步骤之后结束注册。但是，如何在所有网页中保持用户的会话信息。

在许多情况下，使用 cookies 是记忆和跟踪有关用户喜好、购买、佣金以及其他为追求更好的游客体验或网站统计所需信息的最有效的方法。

它是如何工作的

服务器以 cookie 的形式向访客的浏览器发送一些数据。如果浏览器接受了 cookie，则 cookie 会以纯文本记录的形式存储在访客的硬盘上。现在，当访客访问网站上的另一个页面时，会检索 cookie。一旦找到 cookie，服务器就知道存储了什么。

cookie 是一种纯文本的数据记录，带有 5 个可变长度的字段：

- **Expires :** cookie 的过期日期。如果此字段留空，cookie 会在访客退出浏览器时过期。
- **Domain :** 网站的域名。

- **Path :** 设置 cookie 的目录或网页的路径。如果您想从任意的目录或网页检索 cookie，此字段可以留空。
- **Secure :** 如果此字段包含单词 "secure"，那么 cookie 只能通过安全服务器进行检索。如果此字段留空，则不存在该限制。
- **Name=Value :** cookie 以键值对的形式被设置和获取。

设置 Cookies

向浏览器发送 cookies 是非常简单的。这些 cookies 会在 Content-type 字段之前，与 HTTP 头一起被发送。假设您想设置 UserID 和 Password 为 cookies，设置 cookies 的步骤如下所示：

```
#include <iostream>
using namespace std;

int main ()
{

   cout << "Set-Cookie:UserID=XYZ;\r\n";
   cout << "Set-Cookie:Password=XYZ123;\r\n";
   cout << "Set-Cookie:Domain=www.w3cschool.cc;\r\n";
   cout << "Set-Cookie:Path=/perl;\n";
   cout << "Content-type:text/html\r\n\r\n";

   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>CGI 中的 Cookies</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   cout << "设置 cookies" << endl;

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

从这个实例中，我们了解了如何设置 cookies。我们使用 **Set-Cookie** HTTP 头来设置 cookies。

在这里，有一些设置 cookies 的属性是可选的，比如 Expires、Domain 和 Path。值得注意的是，cookies 是在发送行 **"Content-type:text/html\r\n\r\n** 之前被设置的。

编译上面的程序，生成 setcookies.cgi，并尝试使用下面的链接设置 cookies。它会在您的计算机上设置四个 cookies：

```
/cgi-bin/setcookies.cgi
```

获取 Cookies

检索所有设置的 cookies 是非常简单的。cookies 被存储在 CGI 环境变量 HTTP_COOKIE 中，且它们的形式如下：

```
key1=value1;key2=value2;key3=value3....
```

下面的实例演示了如何获取 cookies。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc cgi;
   const_cookie_iterator cci;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>CGI 中的 Cookies</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<table border = \"0\" cellspacing = \"2\">";

   // 获取环境变量
   const CgiEnvironment& env = cgi.getEnvironment();

   for( cci = env.getCookieList().begin();
        cci != env.getCookieList().end();
        ++cci )
   {
      cout << "<tr><td>" << cci->getName() << "</td><td>";
      cout << cci->getValue();
      cout << "</td></tr>\n";
   }
   cout << "</table><\n";

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

现在，编译上面的程序，生成 getcookies.cgi，并尝试使用下面的链接获取您的计算机上所有可用的 cookies：

```
/cgi-bin/getcookies.cgi
```

这会产生一个列表，显示了上一节中设置的四个 cookies 以及您的计算机上所有其他的 cookies：

```
UserID XYZ
Password XYZ123
Domain www.w3cschool.cc
Path /perl
```

文件上传实例

为了上传一个文件，HTML 表单必须把 enctype 属性设置为 **multipart/form-data**。带有文件类型的 input 标签会创建一个 "Browse" 按钮。

```
<html>
<body>
   <form enctype="multipart/form-data"
            action="/cgi-bin/cpp_uploadfile.cgi"
            method="post">
   <p>文件：<input type="file" name="userfile" /></p>
   <p><input type="submit" value="上传" /></p>
   </form>
</body>
</html>
```

这段代码的结果是下面的表单：

```
<form enctype="multipart/form-data" action="/cgi-bin/cpp_uploadfile.cgi" method="post">

文件：&lt;input type="file" name="userfile"&gt;

&lt;input type="reset" value="上传"&gt;

</form>
```

注意：上面的实例已经故意禁用了保存上传的文件在我们的服务器上。您可以在自己的服务器上尝试上面的代码。

下面是用于处理文件上传的脚本 **cpp_uploadfile.cpp**：

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc cgi;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>CGI 中的文件上传</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   // 获取要被上传的文件列表
   const_file_iterator file = cgi.getFile("userfile");
   if(file != cgi.getFiles().end()) {
      // 在 cout 中发送数据类型
      cout << HTTPContentHeader(file->getDataType());
      // 在 cout 中写入内容
      file->writeToStream(cout);
   }
   cout << "<文件上传成功>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

上面的实例是在 **cout** 流中写入内容，但您可以打开文件流，并把上传的文件内容保存在目标位置的某个文件中。

C++ 资源库

C++ STL 教程

在前面的章节中，我们已经学习了 C++ 模板的概念。C++ STL（标准模板库）是一套功能强大的 C++ 模板类，提供了通用的模板类和函数，这些模板类和函数可以实现多种流行和常用的算法和数据结构，如向量、链表、队列、栈。

C++ 标准模板库的核心包括以下三个组件：

组件	描述
容器（Containers）	容器是用来管理某一类对象的集合。C++ 提供了各种不同类型的容器，比如 deque、list、vector、map 等。
算法（Algorithms）	算法作用于容器。它们提供了执行各种操作的方式，包括对容器内容执行初始化、排序、搜索和转换等操作。
迭代器（terators）	迭代器用于遍历对象集合的元素。这些集合可能是容器，也可能是容器的子集。

这三个组件都带有丰富的预定义函数，帮助我们通过简单的方式处理复杂的任务。

下面的程序演示了向量容器（一个 C++ 标准的模板），它与数组十分相似，唯一不同的是，向量在需要扩展大小的时候，会自动处理它自己的存储需求：

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   // 创建一个向量存储 int
   vector<int> vec;
   int i;

   // 显示 vec 的原始大小
   cout << "vector size = " << vec.size() << endl;

   // 推入 5 个值到向量中
   for(i = 0; i < 5; i++){
      vec.push_back(i);
   }

   // 显示 vec 扩展后的大小
   cout << "extended vector size = " << vec.size() << endl;

   // 访问向量中的 5 个值
   for(i = 0; i < 5; i++){
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }

   // 使用迭代器 iterator 访问值
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
      v++;
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
```

关于上面实例中所使用的各种函数，有几点要注意：

- push_back( ) 成员函数在向量的末尾插入值，如果有必要会扩展向量的大小。
- size( ) 函数显示向量的大小。
- begin( ) 函数返回一个指向向量开头的迭代器。
- end( ) 函数返回一个指向向量末尾的迭代器。

C++ 标准库

C++ 标准库可以分为两部分：

- 标准函数库： 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C 语言。
- 面向对象类库： 这个库是类及其相关函数的集合。

C++ 标准库包含了所有的 C 标准库，为了支持类型安全，做了一定的添加和修改。

标准函数库

标准函数库分为以下几类：

- 输入/输出 I/O
- 字符串和字符处理
- 数学
- 时间、日期和本地化
- 动态分配
- 其他
- 宽字符函数

面向对象类库

标准的 C++ 面向对象类库定义了大量支持一些常见操作的类，比如输入/输出 I/O、字符串处理、数值处理。面向对象类库包含以下内容：

- 标准的 C++ I/O 类
- String 类
- 数值类
- STL 容器类
- STL 算法
- STL 函数对象
- STL 迭代器
- STL 分配器
- 本地化库
- 异常处理类
- 杂项支持库

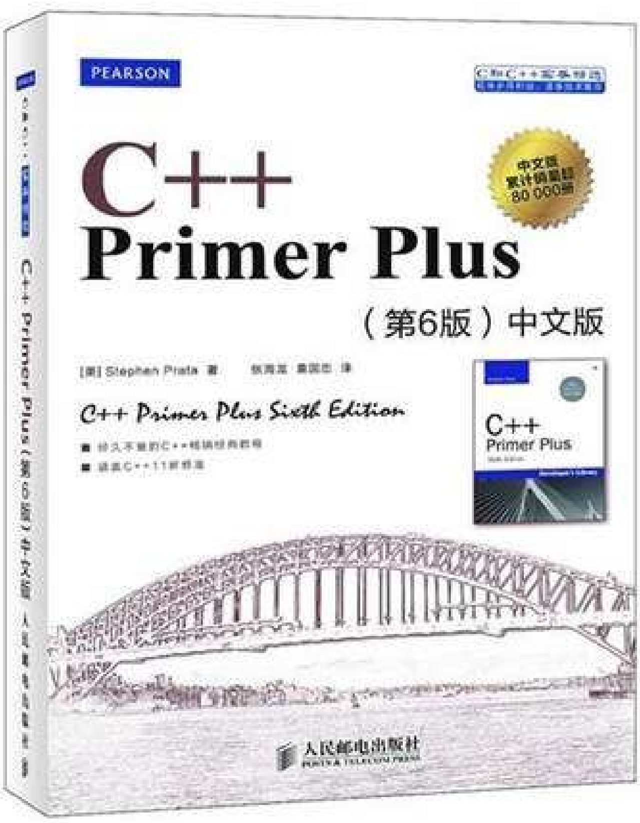
C++ 有用的资源

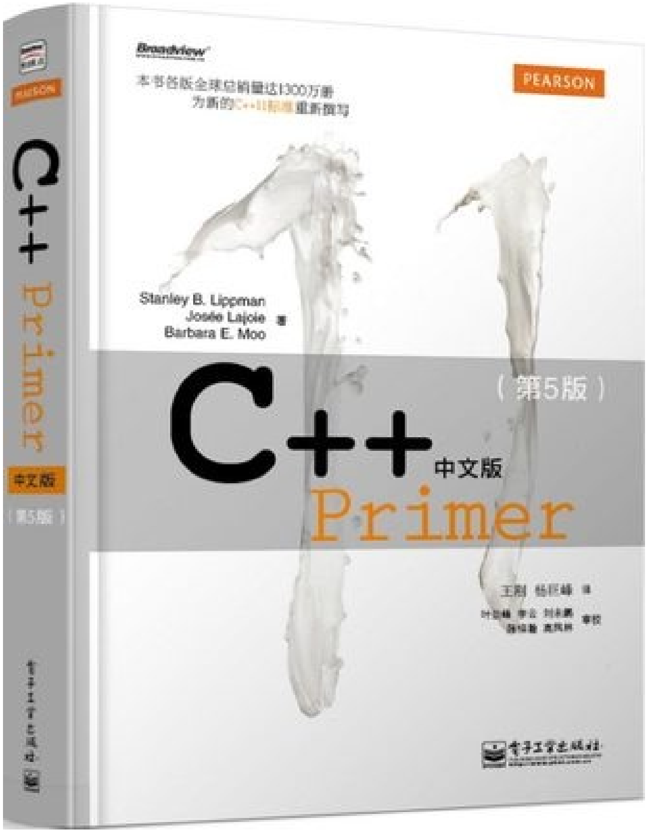
以下资源包含了 C++ 有关的网站、书籍和文章。请使用它们来进一步学习 C++ 的知识。

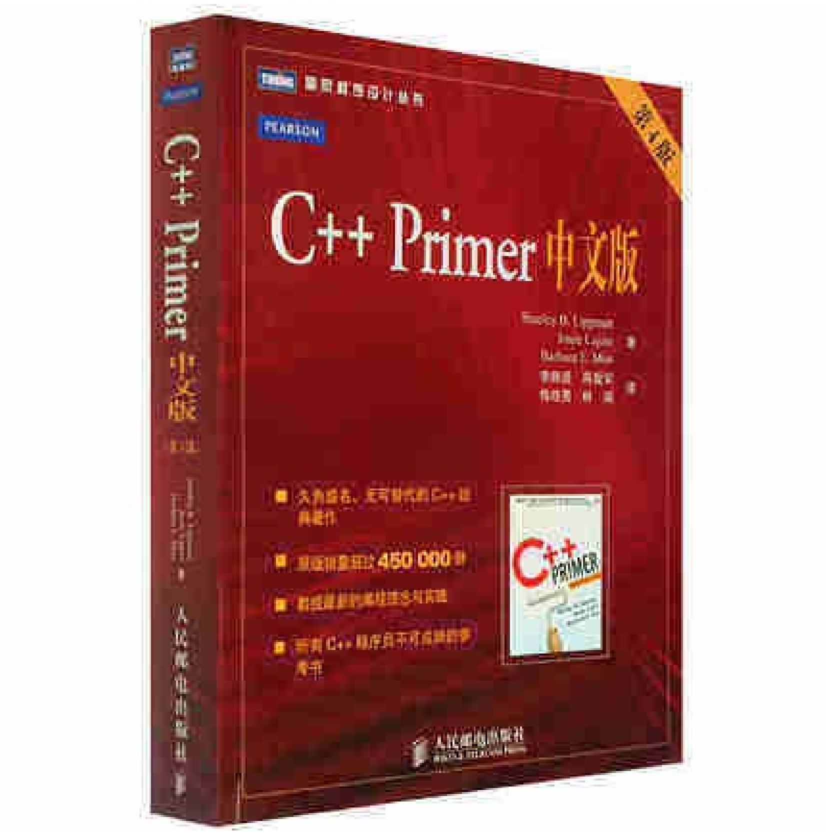
C++ 有用的网站

- C++ Programming Language Tutorials – C++ 编程语言教程。
- C++ Programming – 这本书涵盖了 C++ 语言编程、软件交互设计、C++ 语言的现实生活应用。
- C++ FAQ – C++ 常见问题
- Free Country – Free Country 提供了免费的 C++ 源代码和 C++ 库，这些源代码和库涵盖了压缩、存档、游戏编程、标准模板库和 GUI 编程等 C++ 编程领域。
- C and C++ Users Group – C 和 C++ 的用户团体提供了免费的涵盖各种编程领域 C++ 项目的源代码，包括 AI、动画、编译器、数据库、调试、加密、游戏、图形、GUI、语言工具、系统编程等。

C++ 有用的书籍




Broadview
PEARSON
Stanley B. Lippman
Josée Lajoie
Barbara E. Moo
著
C++
中文版
Primer
（第5版）


PEARSON
C++ Primer 中文版

W3School Lua 教程

作者：W3School

来源：Lua 教程

Lua 简介

Lua 是一种轻量小巧的脚本语言，用标准C语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 是巴西里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro）里的一个研究小组，由Roberto Ierusalimschy、Waldemar Celes 和 Luiz Henrique de Figueiredo所组成并于1993年开发。

设计目的

其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 特性

- 轻量级: 它用标准C语言编写并以源代码形式开放，编译后仅仅一百余K，可以很方便的嵌入别的程序里。
- 可扩展: Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能，Lua可以使用它们，就像是本来就内置的功能一样。
- 其它特性:
    - 支持面向过程(procedure-oriented)编程和函数式编程(functional programming)；
    - 自动内存管理；只提供了一种通用类型的表（table），用它可以实现数组，哈希表，集合，对象；
    - 语言内置模式匹配；闭包(closure)；函数也可以看做一个值；提供多线程（协同进程，并非操作系统所支持的线程）支持；
    - 通过闭包和table可以很方便地支持面向对象编程所需要的一些关键机制，比如数据抽象，虚函数，继承和重载等。

Lua 应用场景

- 游戏开发

- 独立应用脚本
- Web 应用脚本
- 扩展和数据库插件如：MySQL Proxy 和 MySQL WorkBench
- 安全系统，如入侵检测系统

第一个 Lua 程序

接下来我们使用 Lua 来输出"Hello World!"

```
print("Hello World!")
```

运行后，会在屏幕上显示 Hello, world!。

Lua 环境安装

Linux 系统上安装

Linux & Mac上安装 Lua 安装非常简单，只需要下载源码包并在终端解压编译即可，本文使用了5.3.0版本进行安装：

```
curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz tar zxf lua-5.3.0.tar.gz
cd lua-5.3.0 make linux test
make install
```

Mac OS X 系统上安装

```
curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz tar zxf lua-5.3.0.tar.gz
cd lua-5.3.0 make macosx test
make install
```

接下来我们创建一个 helloWorld.lua:

```
print("Hello World!")
```

执行以下命令:

```
$ lua helloWorld
```

输出结果为：

```
Hello  World!
```

Window 系统上安装 Lua

window下你可以使用一个叫"SciTE"的IDE环境来执行lua程序，下载地址为：

- 本站下载地址：http://pan.baidu.com/s/1pJHqF8Z
- Github 下载地址：https://github.com/rjpcomputing/luaforwindows/releases
- Google Code下载地址 : https://code.google.com/p/luaforwindows/downloads/list

双击安装后即可在该环境下编写 Lua 程序并运行。

你也可以使用 Lua 官方推荐的方法使用 LuaDist：http://luadist.org/ <split>123</split>

Lua 基本语法

Lua 学习起来非常简单，我们可以创建第一个 Lua 程序！

第一个 Lua 程序

交互式编程

Lua 提供了交互式编程模式。我们可以在命令行中输入程序并立即查看效果。

Lua 交互式编程模式可以通过命令 lua -i 或 lua 来启用：

```
$ lua -i
$ Lua 5.3.0  Copyright (C) 1994-2015 Lua.org, PUC-Rio
>
```

在命令行中，输入以下命令:

```
> print("Hello World！")
```

接着我们按下回车键，输出结果如下：

```
> print("Hello World！")
Hello World！
>
```

脚本式编程

我们可以将 Lua 程序代码保持到一个以 lua 结尾的文件，并执行，该模式称为脚本式编程，如我们将如下代码存储在名为 hello.lua 的脚本文件中：

```
print("Hello World！")
print("www.w3cschool.cc")
```

使用 lua 名执行以上脚本，输出结果为：

```
$ lua test.lua
Hello World！
www.w3cschool.cc
```

我们也可以将代码修改为如下形式来执行脚本（在开头添加：#!/usr/local/bin/lua）：

```
#!/usr/local/bin/lua

print("Hello World！")
print("www.w3cschool.cc")
```

以上代码中，我们指定了 Lua 的解释器 /usr/local/bin directory。加上 # 号标记解释器会忽略它。接下来我们为脚本添加可执行权限，并执行：

```
./test.lua
Hello World！
www.w3cschool.cc
```

注释

单行注释

两个减号是单行注释:

```
--
```

多行注释

```
--[[
 多行注释
 多行注释
 --]]
```

标示符

Lua 表示符用于定义一个变量，函数获取其他用户定义的项。标示符以一个字母 A 到 Z 或 a 到 z 或下划线 _ 开头后加上0个或多个字母，下划线，数字（0到9）。

最好不要使用下划线加大写字母的标示符，因为Lua的保留字也是这样的。

Lua 不允许使用特殊字符如 @, $, 和 % 来定义标示符。 Lua 是一个区分大小写的编程语言。因此在 Lua 中 W3c 与 w3c 是两个不同的标示符。以下列出了一些正确的标示符：

```
mohd         zara      abc     move_name    a_123
myname50     _temp     j       a23b9        retVal
```

关键词

以下列出了 Lua 的保留关键字。保留关键字不能作为常量或变量或其他用户自定义标示符：

and	break	do	else
elseif	end	false	for
function	if	in	local
nil	not	or	repeat
return	then	true	until
while			

一般约定，以下划线开头连接一串大写字母的名字（比如 _VERSION）被保留用于 Lua 内部全局变量。

全局变量

在默认情况下，变量总是认为是全局的。

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，访问一个没有初始化的全局变量也不会出错，只不过得到的结果是：nil。

```
> print(b)
nil
> b=10
> print(b)
10
>
```

如果你想删除一个全局变量，只需要将变量负值为nil。

```
b = nil
print(b)      --> nil
```

这样变量b就好像从没被使用过一样。换句话说，当且仅当一个变量不等于nil时，这个变量即存在。

Lua 数据类型

Lua是动态类型语言，变量不要类型定义,只需要为变量赋值。 值可以存储在变量中，作为参数传递或结果返回。

Lua中有8个基本类型分别为：nil、boolean、number、string、userdata、function、thread和table。

数据类型	描述
nil	这个最简单，只有值nil属于该类，表示一个无效值（在条件表达式中相当于false）。
boolean	包含两个值：false和true。
number	表示双精度类型的实浮点数
string	字符串由一对双引号或单引号来表示
function	由 C 或 Lua 编写的函数
userdata	表示任意存储在变量中的C数据结构
thread	表示执行的独立线路，用于执行协同程序
table	Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字或者是字符串。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。

我们可以使用type函数测试给定变量或者值的类型：

```
print(type("Hello world"))      --> string
print(type(10.4*3))             --> number
print(type(print))              --> function
print(type(type))               --> function
print(type(true))               --> boolean
print(type(nil))                --> nil
print(type(type(X)))            --> string
```

nil（空）

nil 类型表示一种没有任何有效值，它只有一个值 -- nil，例如打印一个没有赋值的变量，便会输出一个 nil 值：

```
> print(type(a))
nil
>
```

对于全局变量和 table，nil 还有一个"删除"作用，给全局变量或者 table 表里的变量赋一个 nil 值，等同于把它们删掉，执行下面代码就知：

```
tab1 = { key1 = "val1", key2 = "val2", "val3" }
for k, v in pairs(tab1) do
    print(k .. " - " .. v)
end

tab1.key1 = nil
for k, v in pairs(tab1) do
    print(k .. " - " .. v)
end
```

boolean（布尔）

boolean 类型只有两个可选值：true（真） 和 false（假），Lua 把 false 和 nil 看作是"假"，其他的都为"真":

```
print(type(true))
print(type(false))
print(type(nil))

if type(false) or type(nil) then
    print("false and nil are false!")
else
    print("other is true!")
end
```

以上代码执行结果如下：

```
$ lua test.lua
boolean
boolean
nil
false and nil are false!
```

number（数字）

Lua 默认只有一种 number 类型 -- double（双精度）类型（默认类型可以修改 luaconf.h 里的定义），以下几种写法都被看作是 number 类型：

```
print(type(2))
print(type(2.2))
print(type(0.2))
print(type(2e+1))
print(type(0.2e-1))
print(type(7.8263692594256e-06))
```

以上代码执行结果：

```
number
number
number
number
number
number
```

string（字符串）

字符串由一对双引号或单引号来表示。

```
string1 = "this is string1"
string2 = 'this is string2'
```

也可以用 2 个方括号 "[[]]" 来表示"一块"字符串。

```
html = [[
<html>
<head></head>
<body>
    <a href="http://www.w3cschool.cc/">w3cschool菜鸟教程</a>
</body>
</html>
]]
print(html)
```

以下代码执行结果为：

```
<html>
<head></head>
<body>
    <a href="http://www.w3cschool.cc/">w3cschool菜鸟教程</a>
</body>
</html>
```

在对一个数字字符串上进行算术操作时，Lua 会尝试将这个数字字符串转成一个数字:

```
> print("2" + 6)
8.0
> print("2" + "6")
8.0
> print("2 + 6")
2 + 6
> print("-2e2" * "6")
-1200.0
> print("error" + 1)
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
    stdin:1: in main chunk
    [C]: in ?
>
```

以上代码中"error" + 1执行报错了，字符串连接使用的是 .. ，如：

```
> print("a" .. 'b')
ab
> print(157 .. 428)
157428
>
```

使用 # 来计算字符串的长度，放在字符串前面，如下实例：

```
> len = "www.w3cschool.cc"
> print(#len)
16
> print(#"www.w3cschool.cc")
16
>
```

table（表）

在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。也可以在表里添加一些数据，直接初始化表:

```
-- 创建一个空的 table
local tbl1 = {}

-- 直接初始表
local tbl2 = {"apple", "pear", "orange", "grape"}
```

Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字或者是字符串。

```
-- table_test.lua 脚本文件
a = {}
a["key"] = "value"
key = 10
a[key] = 22
a[key] = a[key] + 11
for k, v in pairs(a) do
    print(k .. " : " .. v)
end
```

脚本执行结果为：

```
$ lua table_test.lua
key : value
10 : 33
```

不同于其他语言的数组把 0 作为数组的初始索引，在 Lua 里表的默认初始索引一般以 1 开始。

```
-- table_test2.lua 脚本文件
local tbl = {"apple", "pear", "orange", "grape"}
for key, val in pairs(tbl) do
    print("Key", key)
end
```

脚本执行结果为：

```
$ lua table_test2.lua
Key    1
Key    2
Key    3
Key    4
```

table 不会固定长度大小，有新数据添加时 table 长度会自动增长，没初始的 table 都是 nil。

```
-- table_test3.lua 脚本文件
a3 = {}
for i = 1, 10 do
    a3[i] = i
end
a3["key"] = "val"
print(a3["key"])
print(a3["none"])
```

脚本执行结果为：

```
$ lua table_test3.lua
val
nil
```

function（函数）

在 Lua 中，函数是被看作是"第一类值（First-Class Value）"，函数可以存在变量里:

```
-- function_test.lua 脚本文件
function factorial1(n)
    if n == 0 then
        return 1
    else
        return n * factorial1(n - 1)
    end
end
print(factorial1(5))
factorial2 = factorial1
print(factorial2(5))
```

脚本执行结果为：

```
$ lua function_test.lua
120
120
```

function 可以以匿名函数（anonymous function）的方式通过参数传递:

```
-- function_test2.lua 脚本文件
function anonymous(tab, fun)
    for k, v in pairs(tab) do
        print(fun(k, v))
    end
end
tab = { key1 = "val1", key2 = "val2" }
anonymous(tab, function(key, val)
    return key .. " = " .. val
end)
```

脚本执行结果为：

```
$ lua function_test2.lua
key1 = val1
key2 = val2
```

thread（线程）

在 Lua 里，最主要的线程是协同程序（coroutine）。它跟线程（thread）差不多，拥有自己独立的栈、局部变量和指令指针，可以跟其他协同程序共享全局变量和其他大部分东西。

线程跟协程的区别：线程可以同时多个运行，而协程任意时刻只能运行一个，并且处于运行状态的协程只有被挂起（suspend）时才会暂停。

userdata（自定义类型）

userdata 是一种用户自定义数据，用于表示一种由应用程序或 C/C++ 语言库所创建的类型，可以将任意 C/C++ 的任意数据类型的数据（通常是 struct 和 指针）存储到 Lua 变量中调用。

Lua 变量

变量在使用前，必须在代码中进行声明，即创建该变量。编译程序执行代码之前编译器需要知道如何给语句变量开辟存储区，用于存储变量的值。

Lua 变量有三种类型：全局变量、局部变量、表中的域。

函数外的变量默认为全局变量，除非用 local 显示声明。函数内变量与函数的参数默认为局部变量。

局部变量的作用域为从声明位置开始到所在语句块结束（或者是直到下一个同名局部变量的声明）。

变量的默认值均为 nil。

```
-- test.lua 文件脚本 a =  5  --  全局变量  local b =  5  --  局部变量  function joke() c =  5
```

执行以上实例输出结果为：

```
$ lua test.lua nil  nil  6  6  5  6
```

赋值语句

赋值是改变一个变量的值和改变表域的最基本的方法。

```
a =  "hello"  ..  "world" t.n = t.n +  1
```

Lua可以对多个变量同时赋值，变量列表和值列表的各个元素用逗号分开，赋值语句右边的值会依次赋给左边的变量。

```
a, b =  10,  2*x <--> a=10; b=2*x
```

遇到赋值语句Lua会先计算右边所有的值然后再执行赋值操作，所以我们可以这样进行交换变量的值：

```
x, y = y, x -- swap 'x'  for  'y' a[i], a[j]  = a[j], a[i]  -- swap 'a[i]'  for  'a[i]'
```

当变量个数和值的个数不一致时，Lua会一直以变量个数为基础采取以下策略：

```
a. 变量个数 > 值的个数 按变量个数补足nil b. 变量个数 < 值的个数 多余的值会被忽略
```

例如：

```
a, b, c = 0, 1 print(a,b,c) --> 0 1 nil a, b = a+1, b+1, b+2 -- value of b+2 is
```

上面最后一个例子是一个常见的错误情况，注意：如果要对多个变量赋值必须依次对每个变量赋值。

```
a, b, c = 0, 0, 0 print(a,b,c) --> 0 0 0
```

多值赋值经常用来交换变量，或将函数调用返回给变量：

```
a, b = f()
```

f()返回两个值，第一个赋给a，第二个赋给b。

应该尽可能的使用局部变量，有两个好处：

- 1. 避免命名冲突。
- 2. 访问局部变量的速度比全局变量更快。

索引

对 table 的索引使用方括号 []。Lua 也提供了 . 操作。

```
t[i] t.i -- 当索引为字符串类型时的一种简化写法 gettable_event(t,i) -- 采用索引访问本质上是一个类
```

例如：

```
> site = {} > site["key"] = "www.w3cschool.cc" > print(site["key"]) www.w3cschool.c
```

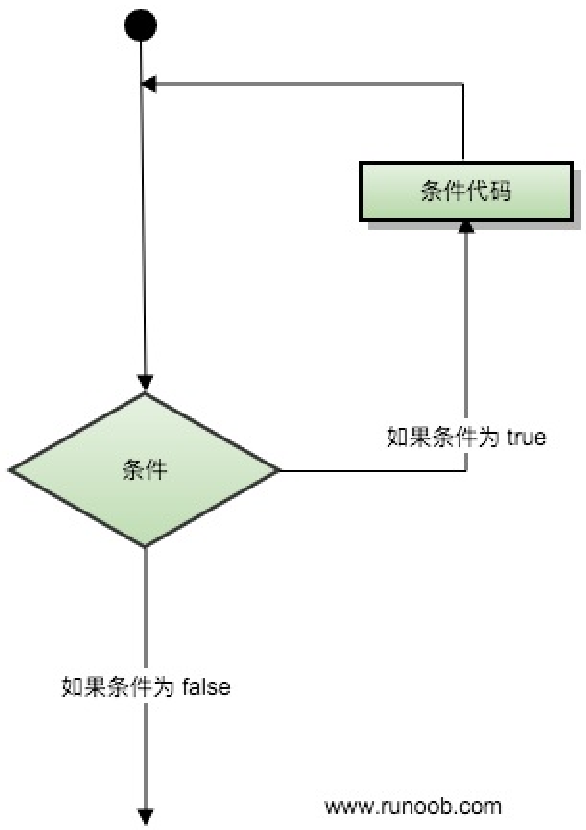
Lua 循环

很多情况下我们需要做一些有规律性的重复操作，因此在程序中就需要重复执行某些语句。

一组被重复执行的语句称之为循环体，能否继续重复，决定循环的终止条件。

循环结构是在一定条件下反复执行某段程序的流程结构，被反复执行的程序被称为循环体。

循环语句是由循环体及循环的终止条件两部分组成的。



Lua 语言提供了以下几种循环处理方式：

循环类型	描述
while 循环	在条件为 true 时，让程序重复地执行某些语句。执行语句前会先检查条件是否为 true。
for 循环	重复执行指定语句，重复次数可在 for 语句中控制。
Lua repeat...until	重复执行循环，直到 指定的条件为真时为止
循环嵌套	可以在循环内嵌套一个或多个循环语句（while、for、do..while）

循环控制语句

循环控制语句用于控制程序的流程， 以实现程序的各种结构方式。

Lua 支持以下循环控制语句：

控制语句	描述
break 语句	退出当前循环或语句，并开始脚本执行紧接着的语句。

无限循环

在循环体中如果条件永远为 true 循环语句就会永远执行下去，以下以 while 循环为例：

```
while( true ) do print("循环将永远执行下去") end
```

Lua while 循环

Lua 编程语言中 while 循环语句在判断条件为 true 时会重复执行循环体语句。

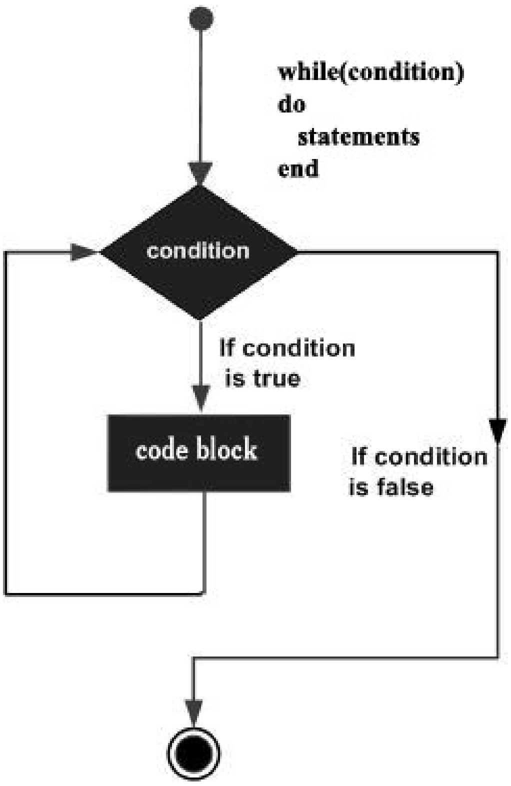
语法

Lua 编程语言中 while 循环语法：

```
while(condition)  do statements end
```

statements(循环体语句**)** 可以是一条或多条语句，**condition(**条件**)** 可以是任意表达式，在 **condition(**条件**)** 为 true 时执行循环体语句。

流程图如下：



在以上流程图中我们可以看出在**condition(**条件**)**为 false 时会跳过当前循环并开始脚本执行紧接着的语句。

实例

以下实例循环输出 a 的值：

```
a=10  while( a <  20  )  do  print("a 的值为:", a) a = a+1  end
```

执行以上代码，输出结果如下：

```
a 的值为:  10 a 的值为:  11 a 的值为:  12 a 的值为:  13 a 的值为:  14 a 的值为:  15 a 的值为:  1
```

Lua for 循环

Lua 编程语言中 for 循环语句可以重复执行指定语句，重复次数可在 for 语句中控制。

Lua 编程语言中 for语句有两大类：：

- 数值for循环
- 泛型for循环

数值for循环

Lua 编程语言中数值for循环语法格式:

```
for  var=exp1,exp2,exp3 do  <执行体>  end
```

var从exp1变化到exp2，每次变化以exp3为步长递增var，并执行一次"执行体"。exp3是可选的，如果不指定，默认为1。

实例

```
for i=1,f(x)  do  print(i)  end  for i=10,1,-1  do  print(i)  end
```

for的三个表达式在循环开始前一次性求值，以后不再进行求值。比如上面的f(x)只会在循环开始前执行一次，其结果用在后面的循环中。

验证如下:

```
#!/usr/local/bin/lua  function f(x)  print("function")  return x*2  end  for i=1,f(5)  do
```

以上实例输出结果为：

```
function  1  2  3  4  5  6  7  8  9  10
```

可以看到 函数f(x)只在循环开始前执行一次。

泛型for循环

泛型for循环通过一个迭代器函数来遍历所有值，类似java中的foreach语句。

Lua 编程语言中泛型for循环语法格式:

```
--打印数组a的所有值  for i,v in ipairs(a)  do  print(v)  end
```

i是数组索引值，v是对应索引的数组元素值。ipairs是Lua提供的一个迭代器函数，用来迭代数组。

实例

循环数组 days：

```
#!/usr/local/bin/lua days =  {"Suanday","Monday","Tuesday","Wednesday","Thursday","Friday
```

以上实例输出结果为：

```
Suanday  Monday  Tuesday  Wednesday  Thursday  Friday  Saturday
```

Lua repeat...until 循环

Lua 编程语言中 repeat...until 循环语句不同于 for 和 while循环，for 和 while循环d的条件语句在当前循环执行开始时判断，而 repeat...until 循环的条件语句在当前循环结束后判断。

语法

Lua 编程语言中 repeat...until 循环语法格式:

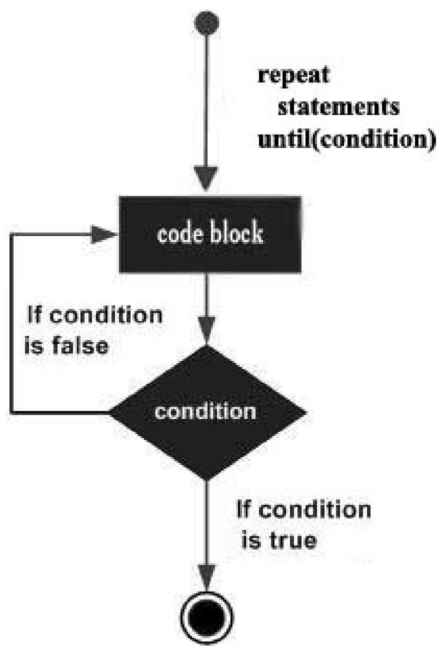
```
repeat
   statements while( condition )
```

repeat...until 是条件后行,所以repeat...until 的循环体里面至少要运行一次。

statements(循环体语句) 可以是一条或多条语句，**condition(条件)** 可以是任意表达式，在 **condition(条件)** 为 true 时执行循环体语句。

在**condition(条件)**为 false 时会跳过当前循环并开始脚本执行紧接着的语句。

Lua repeat...until 循环流程图如下：



实例

```
--[  变量定义  --] a =  10  --[  执行循环  --] repeat print("a的值为:", a) a = a +  1  until(
```

执行以上代码，程序输出结果为：

```
a的值为:  10 a的值为:  11 a的值为:  12 a的值为:  13 a的值为:  14 a的值为:  15
```

Lua 循环嵌套

Lua 编程语言中允许循环中嵌入循环。以下实例演示了 Lua 循环嵌套的应用。

语法

Lua 编程语言中 **for** 循环嵌套语法格式:

```
for init,max/min value, increment do  for init,max/min value, increment do statements end
```

Lua 编程语言中 **while** 循环嵌套语法格式:

```
while(condition)  do  while(condition)  do statements end statements end
```

Lua 编程语言中 **repeat...until** 循环嵌套语法格式:

```
repeat
   statements
   repeat
      statements until( condition )  until( condition )
```

除了以上同类型循环嵌套外，我们还可以使用不同的循环类型来嵌套，如 for 循环体中嵌套 while 循环。

实例

以下实例使用了for循环嵌套:

```
j =2  for i=2,10  do  for j=2,(i/j)  ,  2  do  if(not(i%j))  then  break  end  if(j >  (i
```

以上代码执行结果为：

```
i 的值为：  8 i 的值为：  9 i 的值为：  10
```

Lua break 语句

Lua 编程语言 break 语句插入在循环体中，用于退出当前循环或语句，并开始脚本执行紧接着的语句。

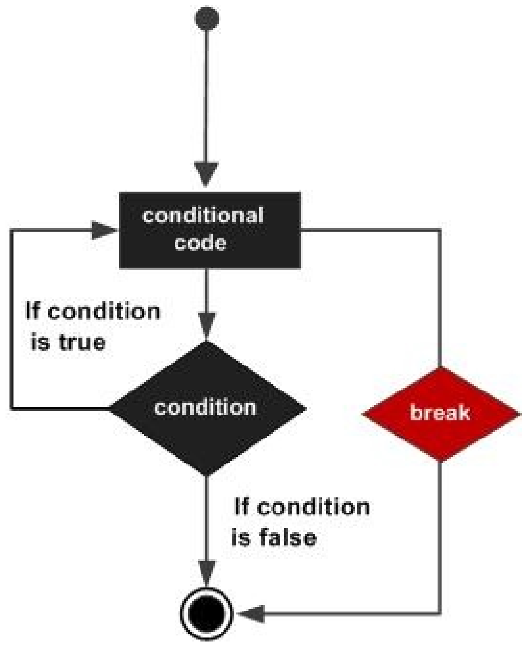
如果你使用循环嵌套，break语句将停止最内层循环的执行，并开始执行的外层的循环语句。

语法

Lua 编程语言中 **break** 语句语法格式:

```
break
```

流程图：



实例

以下实例执行 while 循环，在变量 a 小于 20 时输出 a 的值，并在 a 大于 15 时终止执行循环：

```
--[ 定义变量 --] a = 10 --[ while 循环 --] while( a < 20 ) do print("a 的值为:",
```

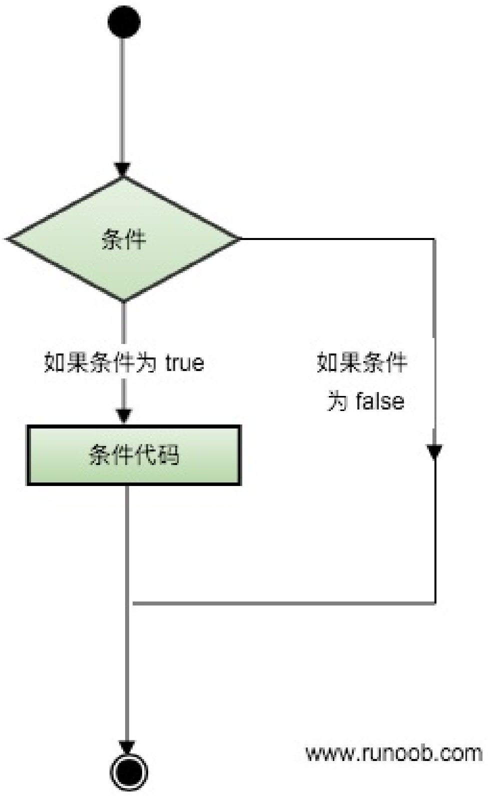
以上代码执行结果如下：

```
a 的值为: 10 a 的值为: 11 a 的值为: 12 a 的值为: 13 a 的值为: 14 a 的值为: 15
```

Lua 流程控制

Lua 编程语言流程控制语句通过程序设定一个或多个条件语句来设定。在条件为 true 时执行指定程序代码，在条件为 false 时执行其他指定代码。

以下是典型的流程控制流程图：



控制结构的条件表达式结果可以是任何值，Lua认为false和nil为假，true 和非nil为真。

要注意的是Lua中 0 为 true：

```
--[ 0 为true ] if(0) then print("0 为真") end
```

以上代码输出结果为：

```
0 为真
```

Lua 提供了以下控制结构语句：

语句	描述
if 语句	**if** 语句 由一个布尔表达式作为条件判断，其后紧跟其他语句组成。
if...else 语句	**if** 语句 可以与 **else** 语句搭配使用, 在 if 条件表达式为 false 时执行 else 语句代码。
if 嵌套语句	你可以在**if** 或 **else if**中使用一个或多个 **if** 或 **else if** 语句 。

Lua if 语句

Lua **if** 语句 由一个布尔表达式作为条件判断，其后紧跟其他语句组成。

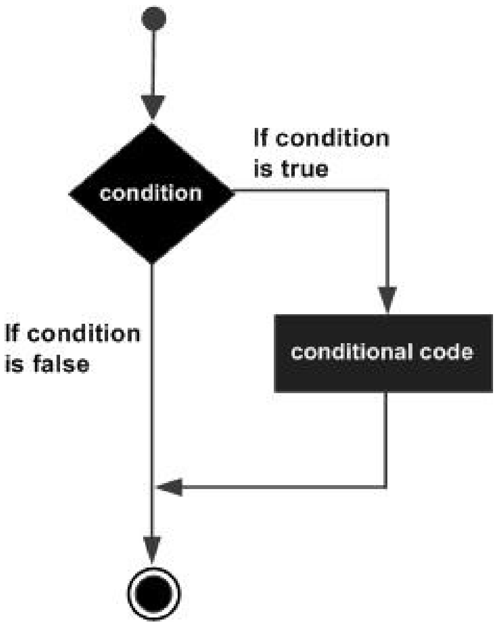
Lua if 语句语法格式如下：

```
if(布尔表达式) then --[ 在布尔表达式为 true 时执行的语句 --] end
```

在布尔表达式为 true 时会if中的代码块会被执行，在布尔表达式为 false 时，紧跟在 if 语句 end 之后的代码会被执行。

Lua认为false和nil为假，true 和非nil为真。要注意的是Lua中 0 为 true。

if 语句流程图如下：



实例

以下实例用于判断变量 a 的值是否小于 20：

```
--[ 定义变量 --] a = 10; --[ 使用 if 语句 --] if( a < 20 ) then --[ if 条件为
```

以上代码执行结果如下：

```
a 小于 20 a 的值为: 10
```

Lua if...else 语句

if...else 语句

Lua if 语句可以与 else 语句搭配使用, 在 if 条件表达式为 false 时执行 else 语句代码块。

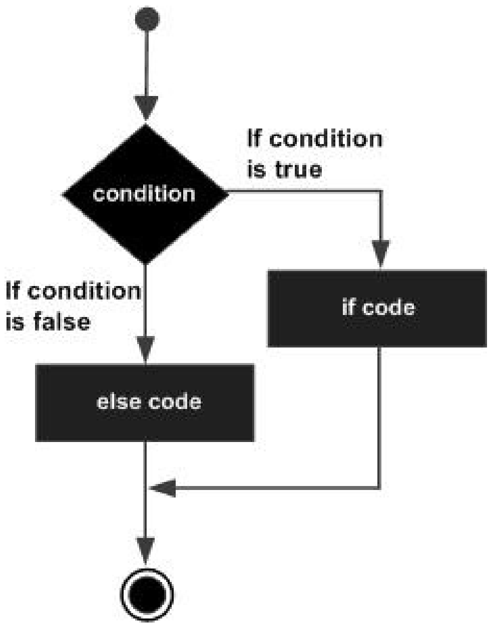
Lua if...else 语句语法格式如下：

```
if(布尔表达式) then --[ 布尔表达式为 true 时执行该语句块 --] else --[ 布尔表达式为 fals
```

在布尔表达式为 true 时会if中的代码块会被执行，在布尔表达式为 false 时，else 的代码块会被执行。

Lua认为false和nil为假，true 和非nil为真。要注意的是Lua中 0 为 true。

if 语句流程图如下：



实例

以下实例用于判断变量 a 的值：

```
--[ 定义变量 --] a = 100; --[ 检查条件 --] if( a < 20 ) then --[ if 条件为 true
```

以上代码执行结果如下：

```
a 大于  20 a 的值为  :  100
```

if...else if...else 语句

Lua if 语句可以与 else if...else 语句搭配使用, 在 if 条件表达式为 false 时执行 else if...else 语句代码块，用于检测多个条件语句。

Lua if...else if...else 语句语法格式如下：

```
if(  布尔表达式  1)  then  --[  在布尔表达式  1  为  true  时执行该语句块  --]  else  if(  布尔表
```

实例

以下实例对变量 a 的值进行判断：

```
--[  定义变量  --] a =  100  --[  检查布尔条件  --]  if( a ==  10  )  then  --[  如果条件为  t
```

以上代码执行结果如下：

```
没有匹配 a 的值 a 的真实值为:  100
```

Lua if 嵌套语句

if...else 语句

Lua if 语句允许嵌套, 这就意味着你可以在一个 if 或 else if 语句中插入其他的 if 或 else if 语句。

Lua if 嵌套语句语法格式如下：

```
if( 布尔表达式 1) then --[ 布尔表达式 1 为 true 时执行该语句块 --] if(布尔表达式 2)
```

你可以用同样的方式嵌套 **else if...else** 语句。

实例

以下实例用于判断变量 a 和 b 的值：

```
--[ 定义变量 --] a = 100; b = 200; --[ 检查条件 --] if( a == 100 ) then --[ if
```

以上代码执行结果如下：

```
a 的值为 100 b 的值为 200 a 的值为 : 100 b 的值为 : 200
```

Lua 函数

在Lua中，函数是对语句和表达式进行抽象的主要方法。既可以用来处理一些特殊的工作，也可以用来计算一些值。

Lua 提供了许多的内建函数，你可以很方便的在程序中调用它们，如print()函数可以将传入的参数打印在控制台上。

Lua 函数主要有两种用途：

- 1.完成指定的任务，这种情况下函数作为调用语句使用；
- 2.计算并返回值，这种情况下函数作为赋值语句的表达式使用。

函数定义

Lua 编程语言函数定义格式如下：

```
optional_function_scope function function_name( argument1, argument2, argument3..., argum
    function_body
    return result_params_comma_separated
end
```

解析：

- **optional_function_scope**

local

- **function_name:**

- **argument1, argument2, argument3..., argumentn:**

- **function_body:**

- **result_params_comma_separated:**

实例

以下实例定义了函数 **max()**，参数为 num1, num2，用于比较两值的大小，并返回最大值：

```
--[[ 函数返回两个值的最大值 --]]
function max(num1, num2)

   if (num1 > num2) then
      result = num1;
   else
      result = num2;
   end

   return result;
end
-- 调用函数
print("两值比较最大值为 ",max(10,4))
print("两值比较最大值为 ",max(5,6))
```

以上代码执行结果为：

```
两值比较最大值为     10
两值比较最大值为     6
```

Lua 中我们可以将函数作为参数传递给函数，如下实例：

```
myprint = function(param)
   print("这是打印函数 -   ##",param,"##")
end

function add(num1,num2,functionPrint)
   result = num1 + num2
   -- 调用传递的函数参数
   functionPrint(result)
end
myprint(10)
-- myprint 函数作为参数传递
add(2,5,myprint)
```

以上代码执行结果为：

```
这是打印函数 -   ##    10    ##
这是打印函数 -   ##    7    ##
```

多返回值

Lua函数可以返回多个结果值，比如string.find，其返回匹配串"开始和结束的下标"（如果不存在匹配串返回nil）。

```
> s, e = string.find("w3cschool菜鸟教程：www.w3cschool.cc", "菜鸟教程")
> print(s,e)
10    21
```

Lua函数中，在return后列出要返回的值得列表即可返回多值，如：

```
function maximum (a)
    local mi = 1             -- 最大值索引
    local m = a[mi]          -- 最大值
    for i,val in ipairs(a) do
       if val > m then
           mi = i
           m = val
       end
    end
    return m, mi
end

print(maximum({8,10,23,12,5}))
```

以上代码执行结果为：

```
23    3
```

可变参数

Lua函数可以接受可变数目的参数，和C语言类似在函数参数列表中使用三点（...) 表示函数有可变的参数。

Lua将函数的参数放在一个叫arg的表中，**#arg** 表示传入参数的个数。

例如，我们计算几个数的平均值：

```
function average(...)
   result = 0
   local arg={...}
   for i,v in ipairs(arg) do
      result = result + v
   end
   print("总共传入 " .. #arg .. " 个数")
   return result/#arg
end

print("平均值为",average(10,5,3,4,5,6))
```

以上代码执行结果为：

```
总共传入 6 个数
平均值为    5.5
```

Lua 运算符

运算符是一个特殊的符号，用于告诉解释器执行特定的数学或逻辑运算。Lua提供了以下几种运算符类型：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 其他运算符

算术运算符

下表列出了 Lua 语言中的常用算术运算符，设定 A 的值为10，B 的值为 20：

操作符	描述	实例
+	加法	A + B 输出结果 30
-	减法	A - B 输出结果 -10
*	乘法	A * B 输出结果 200
/	除法	B / A w输出结果 2
%	取余	B % A 输出结果 0
^	乘幂	A^2 输出结果 100
-	负号	-A 输出结果v -10

实例

我们可以通过以下实例来更加透彻的理解算术运算符的应用：

```
a = 21
b = 10
c = a + b
print("Line 1 - c 的值为 ", c )
c = a - b
print("Line 2 - c 的值为 ", c )
c = a * b
print("Line 3 - c 的值为 ", c )
c = a / b
print("Line 4 - c 的值为 ", c )
c = a % b
print("Line 5 - c 的值为 ", c )
c = a^2
print("Line 6 - c 的值为 ", c )
c = -a
print("Line 7 - c 的值为 ", c )
```

以上程序执行结果为：

```
Line 1 - c 的值为      31
Line 2 - c 的值为      11
Line 3 - c 的值为      210
Line 4 - c 的值为      2.1
Line 5 - c 的值为      1
Line 6 - c 的值为      441
Line 7 - c 的值为      -21
```

关系运算符

下表列出了 Lua 语言中的常用关系运算符，设定 A 的值为10，B 的值为 20：

操作符	描述	实例
==	等于，检测两个值是否相等，相等返回 true，否则返回 false	(A == B) 为 false。
~=	不等于，检测两个值是否相等，相等返回 false，否则返回 true<	(A ~= B) 为 true。
>	大于，如果左边的值大于右边的值，返回 true，否则返回 false	(A > B) 为 false。
<	小于，如果左边的值大于右边的值，返回 false，否则返回 true	(A < B) 为 true。
>=	大于等于，如果左边的值大于等于右边的值，返回 true，否则返回 false	(A >= B) is not true.
<=	小于等于， 如果左边的值小于等于右边的值，返回 true，否则返回 false	(A <= B) is true.

实例

我们可以通过以下实例来更加透彻的理解关系运算符的应用：

```
a = 21
b = 10

if( a == b )
then
   print("Line 1 - a 等于 b" )
else
   print("Line 1 - a 不等于 b" )
end

if( a ~= b )
then
   print("Line 2 - a 不等于 b" )
else
   print("Line 2 - a 等于 b" )
end

if ( a < b )
then
   print("Line 3 - a 小于 b" )
else
   print("Line 3 - a 大于等于 b" )
end

if ( a > b )
then
   print("Line 4 - a 大于 b" )
else
   print("Line 5 - a 小于等于 b" )
end

-- 修改 a 和 b 的值
a = 5
b = 20
if ( a <= b )
then
   print("Line 5 - a 小于等于  b" )
end

if ( b >= a )
then
   print("Line 6 - b 大于等于 a" )
end
```

以上程序执行结果为：

```
Line 1 - a 不等于 b
Line 2 - a 不等于 b
Line 3 - a 大于等于 b
Line 4 - a 大于 b
Line 5 - a 小于等于  b
Line 6 - b 大于等于 a
```

逻辑运算符

下表列出了 Lua 语言中的常用逻辑运算符，设定 A 的值为10，B 的值为 20：

操作符	描述	实例
and	逻辑与操作符。 如果两边的操作都为 true 则条件为 true。	(A and B) 为 false。
or	逻辑或操作符。 如果两边的操作任一一个为 true 则条件为 true。	(A or B) 为 true。
not	逻辑非操作符。与逻辑运算结果相反，如果条件为 true，逻辑非为 false。	!(A and B) 为 false。

实例

我们可以通过以下实例来更加透彻的理解逻辑运算符的应用：

```
a = 5
b = 20

if ( a and b )
then
   print("Line 1 - 条件为 true" )
end

if ( a or b )
then
   print("Line 2 - 条件为 true" )
end

-- 修改 a 和 b 的值
a = 0
b = 10

if ( a and b )
then
   print("Line 3 - 条件为 true" )
else
   print("Line 3 - 条件为 false" )
end

if ( not( a and b) )
then
   print("Line 4 - 条件为 true" )
else
   print("Line 3 - 条件为 false" )
end
```

以上程序执行结果为：

```
Line 1 - 条件为 true
Line 2 - 条件为 true
Line 3 - 条件为 true
Line 3 - 条件为 false
```

其他运算符

下表列出了 Lua 语言中的连接运算符与计算表或字符串长度的运算符：

操作符	描述	实例
..	连接两个字符串	a..b ，其中 a 为 "Hello "， b 为 "World", 输出结果为 "Hello World"。
#	一元运算符，返回字符串或表的长度。	#"Hello" 返回 5

实例

我们可以通过以下实例来更加透彻的理解连接运算符与计算表或字符串长度的运算符的应用：

```
a = "Hello "
b = "World"

print("连接字符串 a 和 b ", a..b )

print("b 字符串长度 ",#b )

print("字符串 Test 长度 ",#"Test" )

print("w3cschool菜鸟教程网址长度 ",#"www.w3cschool.cc" )
```

以上程序执行结果为：

```
连接字符串 a 和 b     Hello World
b 字符串长度     5
字符串 Test 长度     4
w3cschool菜鸟教程网址长度     16
```

运算符优先级

从高到低的顺序：

```
^
not    - (unary)
*      /
+      -
..
<      >      <=     >=     ~=     ==
and
or
```

除了^和..外所有的二元运算符都是左连接的。

```
a+i < b/2+1          <-->       (a+i) < ((b/2)+1)
5+x^2*8              <-->       5+((x^2)*8)
a < y and y <= z     <-->       (a < y) and (y <= z)
-x^2                 <-->       -(x^2)
x^y^z                <-->       x^(y^z)
```

实例

我们可以通过以下实例来更加透彻的了解 Lua 语言运算符的优先级：

```
a = 20
b = 10
c = 15
d = 5

e = (a + b) * c / d;-- ( 30 * 15 ) / 5
print("(a + b) * c / d 运算值为  :",e )

e = ((a + b) * c) / d; -- (30 * 15 ) / 5
print("((a + b) * c) / d 运算值为 :",e )

e = (a + b) * (c / d);-- (30) * (15/5)
print("(a + b) * (c / d) 运算值为 :",e )

e = a + (b * c) / d;  -- 20 + (150/5)
print("a + (b * c) / d 运算值为   :",e )
```

以上程序执行结果为：

```
(a + b) * c / d 运算值为  :    90.0
((a + b) * c) / d 运算值为 :    90.0
(a + b) * (c / d) 运算值为 :    90.0
a + (b * c) / d 运算值为   :    50.0
```

Lua 字符串

字符串或串(String)是由数字、字母、下划线组成的一串字符。

Lua 语言中字符串可以使用以下三种方式来表示：

- 单引号间的一串字符。
- 双引号间的一串字符。
- [[和]]间的一串字符。

以上三种方式的字符串实例如下：

```
string1 =  "Lua"  print("\"字符串 1 是\"",string1) string2 =  'w3cschool.cc'  print("字符串
```

以上代码执行输出结果为：

```
"字符串 1 是"  Lua  字符串  2  是 w3cschool.cc 字符串  3  是  "Lua 教程"
```

转义字符用于表示不能直接显示的字符，比如后退键，回车键，等。如在字符串转换双引号可以使用 "\""。

所有的转义字符和所对应的意义：

转义字符	意义	ASCII码值（十进制）
\a	响铃(BEL)	007
\b	退格(BS)，将当前位置移到前一列	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF)，将当前位置移到下一行开头	010
\r	回车(CR)，将当前位置移到本行开头	013
\t	水平制表(HT) （跳到下一个TAB位置）	009
\v	垂直制表(VT)	011
\\	代表一个反斜线字符''\'	092
\'	代表一个单引号（撇号）字符	039
\"	代表一个双引号字符	034
\0	空字符(NULL)	000
\ddd	1到3位八进制数所代表的任意字符	三位八进制
\xhh	1到2位十六进制所代表的任意字符	二位十六进制

字符串操作

Lua 提供了很多的方法来支持字符串的操作：

方法	用途	
string.upper(argument):	字符串全部转为大写字母。	
string.lower(argument):	字符串全部转为小写字母。	
string.gsub(mainString,findString,replaceString,num)	在字符串中替换,mainString 为要替换的字符串，findString 为被替换的字符，replaceString 要替换的字符，num 替换次数（可以忽略，则全部替换），	`string.gsub`
	在一个指定的目标字符串中	

string.strfind (str, substr, [init, [end]])	搜索指定的内容(第三个参数为索引),返回其具体位置。不存在则返回 nil。	`string.find`
string.reverse(arg)	字符串反转	`string.reve`
string.format(...)	返回一个类似printf的格式化字符串	`string.form`

string.char(arg) 和 string.byte(arg[,int])	char 将整型数字转成字符并连接，byte 转换字符为整数值(可以指定某个字符，默认第一个字符)。	`string.char`
		`string.byte`
		`string.byte`

string.len(arg)	计算字符串长度。	`string.len(`
string.rep(string, n))	返回字符串string的n个拷贝	`string.rep(`
..	链接两个字符串	`print("www.`

字符串大小写转换

以下实例演示了如何对字符串大小写进行转换：

```
string1 =  "Lua";  print(string.upper(string1))  print(string.lower(string1))
```

以上代码执行结果为：

```
LUA
lua
```

字符串查找与反转

以下实例演示了如何对字符串进行查找与反转操作：

```
string  =  "Lua Tutorial"  --  查找字符串  print(string.find(string,"Tutorial")) reversedSt
```

以上代码执行结果为：

```
5  12  新字符串为 lairotuT auL
```

字符串格式化

以下实例演示了如何对字符串进行格式化操作：

```
string1 =  "Lua" string2 =  "Tutorial" number1 =  10 number2 =  20  --  基本字符串格式化  pr
```

以上代码执行结果为：

```
基本格式化  Lua  Tutorial  日期格式化  02/01/2014  0.3333
```

字符与整数相互转换

以下实例演示了字符与整数相互转换：

```
--  字符转换  --  转换第一个字符  print(string.byte("Lua"))  --  转换第三个字符  print(string.b
```

以上代码执行结果为：

```
76  97  97  117  117 a
```

其他常用函数

以下实例演示了其他字符串操作，如计算字符串长度，字符串连接，字符串复制等：

```
string1 =  "www." string2 =  "w3cschool" string3 =  ".cc"  --  使用  ..  进行字符串连接  pri
```

以上代码执行结果为：

```
连接字符串 www.w3cschool.cc 字符串长度  9 w3cschoolw3cschool
```

Lua 数组

数组，就是相同数据类型的元素按一定顺序排列的集合，可以是一维数组和多维数组。

Lua 数组的索引键值可以使用整数表示，数组的大小不是固定的。

一维数组

一维数组是最简单的数组，其逻辑结构是线性表。一维数组可以用for循环出数组中的元素，如下实例：

```
array = {"Lua", "Tutorial"}

for i= 0, 2 do
   print(array[i])
end
```

以上代码执行输出结果为：

```
nil
Lua
Tutorial
```

正如你所看到的，我们可以使用整数索引来访问数组元素，如果知道的索引没有值则返回nil。

在 Lua 索引值是以 1 为起始，但你也可以指定 0 开始。

除此外我们还可以以负数为数组索引值：

```
array = {}

for i= -2, 2 do
   array[i] = i *2
end

for i = -2,2 do
   print(array[i])
end
```

以上代码执行输出结果为：

```
-4
-2
0
2
4
```

多维数组

多维数组即数组中包含数组或一维数组的索引键对应一个数组。

以下是一个三行三列的阵列多维数组：

```
-- 初始化数组
array = {}
for i=1,3 do
   array[i] = {}
      for j=1,3 do
         array[i][j] = i*j
      end
end

-- 访问数组
for i=1,3 do
   for j=1,3 do
      print(array[i][j])
   end
end
```

以上代码执行输出结果为：

```
1
2
3
2
4
6
3
6
9
```

不同索引键的三行三列阵列多维数组：

```
-- 初始化数组
array = {}
maxRows = 3
maxColumns = 3
for row=1,maxRows do
   for col=1,maxColumns do
      array[row*maxColumns +col] = row*col
   end
end

-- 访问数组
for row=1,maxRows do
   for col=1,maxColumns do
      print(array[row*maxColumns +col])
   end
end
```

以上代码执行输出结果为：

```
1
2
3
2
4
6
3
6
9
```

正如你所看到的，以上的实例中，数组设定了指定的索引值，这样可以避免出现 nil 值，有利于节省内存空间。

Lua 迭代器

迭代器（iterator）是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址

在Lua中迭代器是一种支持指针类型的结构，它可以遍历集合的每一个元素。

泛型 for 迭代器

泛型 for 在自己内部保存迭代函数，实际上它保存三个值：迭代函数、状态常量、控制变量。

泛型 for 迭代器提供了集合的 key/value 对，语法格式如下：

```
for k, v in pairs(t) do
    print(k, v)
end
```

上面代码中，k, v为变量列表；pair(t)为表达式列表。

查看以下实例:

```
array = {"Lua", "Tutorial"}

for key,value in ipairs(array)
do
   print(key, value)
end
```

以上代码执行输出结果为：

```
1  Lua
2  Tutorial
```

以上实例中我们使用了 Lua 默认提供的迭代函数 ipairs。

下面我们看看范性for的执行过程：

- 首先，初始化，计算in后面表达式的值，表达式应该返回范性for需要的三个值：迭代函数、状态常量、控制变量；与多值赋值一样，如果表达式返回的结果个数不足三个会自动用nil补足，多出部分会被忽略。
- 第二，将状态常量和控制变量作为参数调用迭代函数（注意：对于for结构来说，状态常量没有用处，仅仅在初始化时获取他的值并传递给迭代函数）。
- 第三，将迭代函数返回的值赋给变量列表。
- 第四，如果返回的第一个值为nil循环结束，否则执行循环体。

- 第五，回到第二步再次调用迭代函数

。在Lua中我们常常使用函数来描述迭代器，每次调用该函数就返回集合的下一个元素。Lua的迭代器包含以下两种类型：

- 无状态的迭代器
- 多状态的迭代器

无状态的迭代器

无状态的迭代器是指不保留任何状态的迭代器，因此在循环中我们可以利用无状态迭代器避免创建闭包花费额外的代价。

每一次迭代，迭代函数都是用两个变量（状态常量和控制变量）的值作为参数被调用，一个无状态的迭代器只利用这两个值可以获取下一个元素。

这种无状态迭代器的典型的简单的例子是ipairs，他遍历数组的每一个元素。

以下实例我们使用了一个简单的函数来实现迭代器，实现 数字 n 的平方：

```
function square(iteratorMaxCount,currentNumber)
   if currentNumber<iteratorMaxCount
   then
      currentNumber = currentNumber+1
   return currentNumber, currentNumber*currentNumber
   end
end

for i,n in square,3,0
do
   print(i,n)
end
```

以上实例输出结果为：

```
1    1
2    4
3    9
```

迭代的状态包括被遍历的表（循环过程中不会改变的状态常量）和当前的索引下标（控制变量），ipairs和迭代函数都很简单，我们在Lua中可以这样实现：

```
function iter (a, i)
    i = i + 1
    local v = a[i]
    if v then
       return i, v
    end
end

function ipairs (a)
    return iter, a, 0
end
```

当Lua调用ipairs(a)开始循环时，他获取三个值：迭代函数iter、状态常量a、控制变量初始值0；然后Lua调用iter(a,0)返回1,a[1]（除非a[1]=nil）；第二次迭代调用iter(a,1)返回2,a[2]……直到第一个非nil元素。

多状态的迭代器

很多情况下，迭代器需要保存多个状态信息而不是简单的状态常量和控制变量，最简单的方法是使用闭包，还有一种方法就是将所有的状态信息封装到table内，将table作为迭代器的状态常量，因为这种情况下可以将所有的信息存放在table内，所以迭代函数通常不需要第二个参数。

以下实例我们创建了自己的迭代器：

```
array = {"Lua", "Tutorial"}

function elementIterator (collection)
   local index = 0
   local count = #collection
   -- 闭包函数
   return function ()
      index = index + 1
      if index <= count
      then
         --  返回迭代器的当前元素
         return collection[index]
      end
   end
end

for element in elementIterator(array)
do
   print(element)
end
```

以上实例输出结果为：

```
Lua
Tutorial
```

以上实例中我们可以看到，elementIterator 内使用了闭包函数，实现计算集合大小并输出各个元素。

Lua table(表)

table 是 Lua 的一种数据结构用来帮助我们创建不同的数据类型，如：数字、字典等。

Lua table 使用关联型数组，你可以用任意类型的值来作数组的索引，但这个值不能是 nil。

Lua table 是不固定大小的，你可以根据自己需要进行扩容。

Lua也是通过table来解决模块（module）、包（package）和对象（Object）的。 例如string.format表示使用"format"来索引table string。

table(表)的构造

构造器是创建和初始化表的表达式。表是Lua特有的功能强大的东西。最简单的构造函数是{}，用来创建一个空表。可以直接初始化数组:

```
-- 初始化表
mytable = {}

-- 指定值
mytable[1]= "Lua"

-- 移除引用
mytable = nil
-- lua 垃圾回收会释放内存
```

当我们为 table a 并设置元素，然后将 a 赋值给 b，则 a 与 b 都指向同一个内存。如果 a 设置为 nil ，则 b 同样能访问 table 的元素。如果没有指定的变量指向a，Lua的垃圾回收机制会清理相对应的内存。

以下实例演示了以上的描述情况：

```
-- 简单的 table
mytable = {}
print("mytable 的类型是 ",type(mytable))

mytable[1]= "Lua"
mytable["wow"] = "修改前"
print("mytable 索引为 1 的元素是 ", mytable[1])
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

-- alternatetable和mytable的是指同一个 table
alternatetable = mytable

print("alternatetable 索引为 1 的元素是 ", alternatetable[1])
print("mytable 索引为 wow 的元素是 ", alternatetable["wow"])

alternatetable["wow"] = "修改后"

print("mytable 索引为 wow 的元素是 ", mytable["wow"])

-- 释放变量
alternatetable = nil
print("alternatetable 是 ", alternatetable)

-- mytable 仍然可以访问
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

mytable = nil
print("mytable 是 ", mytable)
```

以上代码执行结果为：

```
mytable 的类型是      table
mytable 索引为 1 的元素是      Lua
mytable 索引为 wow 的元素是      修改前
alternatetable 索引为 1 的元素是      Lua
mytable 索引为 wow 的元素是      修改前
mytable 索引为 wow 的元素是      修改后
alternatetable 是      nil
mytable 索引为 wow 的元素是      修改后
mytable 是      nil
```

Table 操作

以下列出了 Table 操作常用的方法：

方法	用途
table.concat (table [, step [, start [, end]]]):	concat是concatenate(连锁, 连接)的缩写. table.concat()函数列出参数中指定table的数组部分从start位置到end位置的所有元素, 元素间以指定的分隔符(sep)隔开。
table.insert (table, [pos,] value):	在table的数组部分指定位置(pos)插入值为value的一个元素. pos参数可选, 默认为数组部分末尾.
table.maxn (table)	指定table中所有正数key值中最大的key值. 如果不存在key值为正数的元素, 则返回0。(**Lua5.2**之后该方法已经不存在了,本文使用了自定义函数实现)
table.remove (table [, pos])	返回table数组部分位于pos位置的元素. 其后的元素会被前移. pos参数可选, 默认为table长度, 即从最后一个元素删起。
table.sort (table [, comp])	对给定的table进行升序排序。

接下来我们来看下这几个方法的实例。

Table 连接

我们可以使用 concat() 方法来连接两个 table:

```
fruits = {"banana","orange","apple"}
-- 返回 table 连接后的字符串
print("连接后的字符串 ",table.concat(fruits))

-- 指定连接字符
print("连接后的字符串 ",table.concat(fruits,", "))

-- 指定索引来连接 table
print("连接后的字符串 ",table.concat(fruits,", ", 2,3))
```

执行以上代码输出结果为：

```
连接后的字符串     bananaorangeapple
连接后的字符串     banana, orange, apple
连接后的字符串     orange, apple
```

插入和移除

以下实例演示了 table 的插入和移除操作:

```
fruits = {"banana","orange","apple"}

-- 在末尾插入
table.insert(fruits,"mango")
print("索引为 4 的元素为 ",fruits[4])

-- 在索引为 2 的键处插入
table.insert(fruits,2,"grapes")
print("索引为 2 的元素为 ",fruits[2])

print("最后一个元素为 ",fruits[5])
table.remove(fruits)
print("移除后最后一个元素为 ",fruits[5])
```

执行以上代码输出结果为：

```
索引为 4 的元素为 	mango
索引为 2 的元素为 	grapes
最后一个元素为 	mango
移除后最后一个元素为 	nil
```

Table 排序

以下实例演示了 sort() 方法的使用，用于对 Table 进行排序：

```
fruits = {"banana","orange","apple","grapes"}
print("排序前")
for k,v in ipairs(fruits) do
    print(k,v)
end

table.sort(fruits)
print("排序后")
for k,v in ipairs(fruits) do
    print(k,v)
end
```

执行以上代码输出结果为：

```
排序前
1	banana
2	orange
3	apple
4	grapes
排序后
1	apple
2	banana
3	grapes
4	orange
```

Table 最大值

table.maxn 在 Lua5.2 之后该方法已经不存在了，我们定义了 table_maxn 方法来实现。

以下实例演示了如何获取 table 中的最大值：

```
function table_maxn(t)
    local mn = 0
    for k, v in pairs(t) do
        if mn < k then
            mn = k
        end
    end
    return mn
end
tbl = {[1] = "a", [2] = "b", [3] = "c", [26] = "z"}
print("tbl 长度 ", #tbl)
print("tbl 最大值 ", table_maxn(tbl))
```

执行以上代码输出结果为：

```
tbl 长度     3
tbl 最大值     26
```

Lua 模块与包

模块类似于一个封装库，从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

Lua 的模块是由变量、函数等已知元素组成的 table，因此创建一个模块很简单，就是创建一个 table，然后把需要导出的常量、函数放入其中，最后返回这个 table 就行。以下为创建自定义模块 module.lua，文件代码格式如下：

```
-- 文件名为 module.lua
-- 定义一个名为 module 的模块
module = {}

-- 定义一个常量
module.constant = "这是一个常量"

-- 定义一个函数
function module.func1()
    io.write("这是一个公有函数！\n")
end

local function func2()
    print("这是一个私有函数！")
end

function module.func3()
    func2()
end

return module
```

由上可知，模块的结构就是一个 table 的结构，因此可以像操作调用 table 里的元素那样来操作调用模块里的常量或函数。

上面的 func2 声明为程序块的局部变量，即表示一个私有函数，因此是不能从外部访问模块里的这个私有函数，必须通过模块里的公有函数来调用.

require 函数

Lua提供了一个名为require的函数用来加载模块。要加载一个模块，只需要简单地调用就可以了。例如：

```
require("<模块名>")
```

或者

```
require "<模块名>"
```

执行 require 后会返回一个由模块常量或函数组成的 table，并且还会定义一个包含该 table 的全局变量。

```
-- test_module.php 文件
-- module 模块为上文提到到 module.lua
require("module")

print(module.constant)

module.func3()
```

以上代码执行结果为：

```
这是一个常量
这是一个私有函数！
```

或者给加载的模块定义一个别名变量，方便调用：

```
-- test_module2.php 文件
-- module 模块为上文提到到 module.lua
-- 别名变量 m
local m = require("module")

print(m.constant)

m.func3()
```

以上代码执行结果为：

```
这是一个常量
这是一个私有函数！
```

加载机制

对于自定义的模块，模块文件不是放在哪个文件目录都行，函数 require 有它自己的文件路径加载策略，它会尝试从 Lua 文件或 C 程序库中加载模块。

require 用于搜索 Lua 文件的路径是存放在全局变量 package.path 中，当 Lua 启动后，会以环境变量 LUA_PATH 的值来初始这个环境变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。

当然，如果没有 LUA_PATH 这个环境变量，也可以自定义设置，在当前用户根目录下打开 .profile 文件（没有则创建，打开 .bashrc 文件也可以），例如把 "~/lua/" 路径加入 LUA_PATH 环境变量里：

```
#LUA_PATH
export LUA_PATH="~/lua/?.lua;;"
```

文件路径以 ";" 号分隔，最后的 2 个 ";;" 表示新加的路径后面加上原来的默认路径。

接着，更新环境变量参数，使之立即生效。

```
source ~/.profile
```

这时假设 package.path 的值是：

```
/Users/dengjoe/lua/?.lua;./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/
```

那么调用 require("module") 时就会尝试打开以下文件目录去搜索目标。

```
/Users/dengjoe/lua/module.lua;
./module.lua
/usr/local/share/lua/5.1/module.lua
/usr/local/share/lua/5.1/module/init.lua
/usr/local/lib/lua/5.1/module.lua
/usr/local/lib/lua/5.1/module/init.lua
```

如果找过目标文件，则会调用 package.loadfile 来加载模块。否则，就会去找 C 程序库。

搜索的文件路径是从全局变量 package.cpath 获取，而这个变量则是通过环境变量 LUA_CPATH 来初始。

搜索的策略跟上面的一样，只不过现在换成搜索的是 so 或 dll 类型的文件。如果找得到，那么 require 就会通过 package.loadlib 来加载它。

C 包

Lua和C是很容易结合的，使用C为Lua写包。

与Lua中写包不同，C包在使用以前必须首先加载并连接，在大多数系统中最容易的实现方式是通过动态连接库机制。

Lua在一个叫loadlib的函数内提供了所有的动态连接的功能。这个函数有两个参数:库的绝对路径和初始化函数。所以典型的调用的例子如下:

```
local path = "/usr/local/lua/lib/libluasocket.so"
local f = loadlib(path, "luaopen_socket")
```

loadlib函数加载指定的库并且连接到Lua，然而它并不打开库（也就是说没有调用初始化函数），反之他返回初始化函数作为Lua的一个函数，这样我们就可以直接在Lua中调用他。

如果加载动态库或者查找初始化函数时出错，loadlib将返回nil和错误信息。我们可以修改前面一段代码，使其检测错误然后调用初始化函数：

```
local path = "/usr/local/lua/lib/libluasocket.so"
-- 或者 path = "C:\\windows\\luasocket.dll"，这是 Window 平台下
local f = assert(loadlib(path, "luaopen_socket"))
f()  -- 真正打开库
```

一般情况下我们期望二进制的发布库包含一个与前面代码段相似的stub文件，安装二进制库的时候可以随便放在某个目录，只需要修改stub文件对应二进制库的实际路径即可。

将stub文件所在的目录加入到LUA_PATH，这样设定后就可以使用require函数加载C库了。

Lua 元表(Metatable)

在 Lua table 中我们可以访问对应的key来得到value值，但是却无法对两个 table 进行操作。

因此 Lua 提供了元表(Metatable)，允许我们改变table的行为，每个行为关联了对应的元方法。

例如，使用元表我们可以定义Lua如何计算两个table的相加操作a+b。

当Lua试图对两个表进行相加时，先检查两者之一是否有元表，之后检查是否有一个叫**"__add"**的字段，若找到，则调用对应的值。**"__add"**等即时字段，其对应的值（往往是一个函数或是table）就是"元方法"。

有两个很重要的函数来处理元表：

- **setmetatable(table,metatable):** 对指定table设置元表(metatable)，如果元表(metatable)中存在__metatable键值，setmetatable会失败 。
- **getmetatable(table):** 返回对象的元表(metatable)。

以下实例演示了如何对指定的表设置元表：

```
mytable = {}
mymetatable = {}
setmetatable(mytable,mymetatable)
```

以上代码也可以直接写成一行：

```
mytable = setmetatable({},{})
```

__index 元方法

__index 元方法查看表中元素是否存在，如果不存在，返回结果为**nil**；如果存在则由__index 返回结果。

```
mytable = setmetatable({key1 = "value1"}, {
  __index = function(mytable, key)
    if key == "key2" then
      return "metatablevalue"
    else
      return mytable[key]
    end
  end
})

print(mytable.key1,mytable.key2)
```

实例输出结果为：

```
value1    metatablevalue
```

实例解析：

- mytable 表赋值为 **{key1 = "value1"}**。
- mytable 设置了元表，元方法为 __index。
- 在mytable表中查找 key1，如果找到，返回该元素，找不到则继续。
- 在mytable表中查找 key2，如果找到，返回该元素，找不到则继续。
- 判断元表有没有**index**方法，如果index方法是一个函数，则调用该函数。
- 元方法中查看是否传入 "key2" 键的参数（mytable.key2已设置），如果传入 "key2" 参数返回 "metatablevalue"，否则返回 mytable 对应的键值。

我们可以将以上代码简单写成：

```
mytable = setmetatable({key1 = "value1"}, { __index = { key2 = "metatablevalue" } })
print(mytable.key1,mytable.key2)
```

__newindex 元方法

newindex 元方法用来对表更新，index则用来对表访问 。

当你给表的一个缺少的索引赋值，解释器就会查找__newindex 元方法：如果存在则调用这个函数而不进行赋值操作。

以下实例演示了 __newindex 元方法的应用：

```
mymetatable = {}
mytable = setmetatable({key1 = "value1"}, { __newindex = mymetatable })

print(mytable.key1)

mytable.newkey = "新值2"
print(mytable.newkey,mymetatable.newkey)

mytable.key1 = "新值1"
print(mytable.key1,mymetatable.newkey1)
```

以上实例执行输出结果为：

```
value1
nil    新值2
新值1    nil
```

以上实例中表设置了元方法 **newindex**，在对新索引键（**newkey**）赋值时（**mytable.newkey = "新值2"**），会调用元方法，而不进行赋值。而如果对已存在的索引键（**key1**），则会进行赋值，而不调用元方法 newindex。

以下实例使用了 rawset 函数来更新表：

```
mytable = setmetatable({key1 = "value1"}, {
  __newindex = function(mytable, key, value)
        rawset(mytable, key, "\""..value.."\"")

  end
})

mytable.key1 = "new value"
mytable.key2 = 4

print(mytable.key1,mytable.key2)
```

以上实例执行输出结果为：

```
new value    "4"
```

为表添加操作符

以下实例演示了两表相加操作：

```
-- 计算表中最大值，table.maxn在Lua5.2以上版本中已无法使用
-- 自定义计算表中最大值函数 table_maxn
function table_maxn(t)
    local mn = 0
    for k, v in pairs(t) do
        if mn < k then
            mn = k
        end
    end
    return mn
end

-- 两表相加操作
mytable = setmetatable({ 1, 2, 3 }, {
  __add = function(mytable, newtable)
    for i = 1, table_maxn(newtable) do
      table.insert(mytable, table_maxn(mytable)+1,newtable[i])
    end
    return mytable
  end
})

secondtable = {4,5,6}

mytable = mytable + secondtable
    for k,v in ipairs(mytable) do
print(k,v)
end
```

以上实例执行输出结果为：

```
1	1
2	2
3	3
4	4
5	5
6	6
```

__add 键包含在元表中，并进行相加操作。 表中对应的操作列表如下：

模式	描述
__add	对应的运算符 '+'.
__sub	对应的运算符 '-'.
__mul	对应的运算符 '*'.
__div	对应的运算符 '/'.
__mod	对应的运算符 '%'.
__unm	对应的运算符 '-'.
__concat	对应的运算符 '..'.
__eq	对应的运算符 '=='.
__lt	对应的运算符 '<'.
__le	对应的运算符 '<='.

__call 元方法

__call 元方法在 Lua 调用一个值时调用。以下实例演示了计算表中元素的和：

```
-- 计算表中最大值，table.maxn在Lua5.2以上版本中已无法使用
-- 自定义计算表中最大值函数 table_maxn
function table_maxn(t)
    local mn = 0
    for k, v in pairs(t) do
        if mn < k then
            mn = k
        end
    end
    return mn
end

-- 定义元方法__call
mytable = setmetatable({10}, {
  __call = function(mytable, newtable)
    sum = 0
    for i = 1, table_maxn(mytable) do
        sum = sum + mytable[i]
    end
    for i = 1, table_maxn(newtable) do
        sum = sum + newtable[i]
    end
    return sum
  end
})
newtable = {10,20,30}
print(mytable(newtable))
```

以上实例执行输出结果为：

```
70
```

__tostring 元方法

__tostring 元方法用于修改表的输出行为。以下实例我们自定义了表的输出内容：

```
mytable = setmetatable({ 10, 20, 30 }, {
  __tostring = function(mytable)
    sum = 0
    for k, v in pairs(mytable) do
        sum = sum + v
    end
    return "表所有元素的和为 " .. sum
  end
})
print(mytable)
```

以上实例执行输出结果为：

```
表所有元素的和为 60
```

从本文中我们可以知道元表可以很好的简化我们的代码功能，所以了解 Lua 的元表，可以让我们写出更加简单优秀的 Lua 代码。

Lua 协同程序(coroutine)

什么是协同(coroutine)？

Lua 协同程序(coroutine)与线程比较类似：拥有独立的堆栈，独立的局部变量，独立的指令指针，同时又与其它协同程序共享全局变量和其它大部分东西。

协同是非常强大的功能，但是用起来也很复杂。

线程和协同程序区别

线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作的运行。

在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只有在明确的被要求挂起的时候才会被挂起。

协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同。

基本语法

方法	描述
coroutine.create()	创建coroutine，返回coroutine， 参数是一个函数，当和resume配合使用的时候就唤醒函数调用
coroutine.resume()	重启coroutine，和create配合使用
coroutine.yield()	挂起coroutine，将coroutine设置为挂起状态，这个和resume配合使用能有很多有用的效果
coroutine.status()	查看coroutine的状态 注：coroutine的状态有三种：dead，suspend，running，具体什么时候有这样的状态请参考下面的程序
coroutine.wrap（）	创建coroutine，返回一个函数，一旦你调用这个函数，就进入coroutine，和create功能重复
coroutine.running()	返回正在跑的coroutine，一个coroutine就是一个线程，当使用running的时候，就是返回一个corouting的线程号

以下实例演示了以上各个方法的用法：

```
-- coroutine_test.lua 文件
co = coroutine.create(
    function(i)
        print(i);
    end
)

coroutine.resume(co, 1)   -- 1
print(coroutine.status(co))  -- dead

print("----------")

co = coroutine.wrap(
    function(i)
        print(i);
    end
)

co(1)

print("----------")

co2 = coroutine.create(
    function()
        for i=1,10 do
            print(i)
            if i == 3 then
                print(coroutine.status(co2))  --running
                print(coroutine.running()) --thread:XXXXXX
            end
            coroutine.yield()
        end
    end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2))   -- suspended
print(coroutine.running())   --nil

print("----------")
```

以上实例执行输出结果为：

```
1
dead
----------
1
----------
1
2
3
running
thread: 0x7fb801c05868    false
suspended
thread: 0x7fb801c04c88    true
----------
```

coroutine.running就可以看出来,coroutine在底层实现就是一个线程。

当create一个coroutine的时候就是在新线程中注册了一个事件。

当使用resume触发事件的时候，create的coroutine函数就被执行了，当遇到yield的时候就代表挂起当前线程，等候再次resume触发事件。

接下来我们分析一个更详细的实例：

```
function foo (a)
    print("foo 函数输出", a)
    return coroutine.yield(2 * a) -- 返回  2*a 的值
end

co = coroutine.create(function (a , b)
    print("第一次协同程序执行输出", a, b) -- co-body 1 10
    local r = foo(a + 1)

    print("第二次协同程序执行输出", r)
    local r, s = coroutine.yield(a + b, a - b)  -- a，b的值为第一次调用协同程序时传入

    print("第三次协同程序执行输出", r, s)
    return b, "结束协同程序"                   -- b的值为第二次调用协同程序时传入
end)

print("main", coroutine.resume(co, 1, 10)) -- true, 4
print("--分割线----")
print("main", coroutine.resume(co, "r")) -- true 11 -9
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- true 10 end
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- cannot resume dead coroutine
print("---分割线---")
```

以上实例执行输出结果为：

```
第一次协同程序执行输出    1    10
foo 函数输出    2
main    true    4
--分割线----
第二次协同程序执行输出    r
main    true    11    -9
---分割线---
第三次协同程序执行输出    x    y
main    true    10    结束协同程序
---分割线---
main    false    cannot resume dead coroutine
---分割线---
```

以上实例接下如下：

- 调用resume，将协同程序唤醒,resume操作成功返回true，否则返回false；
- 协同程序运行；
- 运行到yield语句；
- yield挂起协同程序，第一次resume返回；（注意：此处yield返回，参数是resume的参数）
- 第二次resume，再次唤醒协同程序；（注意：此处resume的参数中，除了第一个参数，剩下的参数将作为yield的参数）
- yield返回；
- 协同程序继续运行；

- 如果使用的协同程序继续运行完成后继续调用 resumev方法则输出：cannot resume dead coroutine

resume和yield的配合强大之处在于，resume处于主程中，它将外部状态（数据）传入到协同程序内部；而yield则将内部的状态（数据）返回到主程中。

生产者-消费者问题

现在我就使用Lua的协同程序来完成生产者-消费者这一经典问题。

```
local newProductor

function productor()
     local i = 0
     while true do
          i = i + 1
          send(i)     -- 将生产的物品发送给消费者
     end
end

function consumer()
     while true do
          local i = receive()     -- 从生产者那里得到物品
          print(i)
     end
end

function receive()
     local status, value = coroutine.resume(newProductor)
     return value
end

function send(x)
     coroutine.yield(x)     -- x表示需要发送的值，值返回以后，就挂起该协同程序
end

-- 启动程序
newProductor = coroutine.create(productor)
consumer()
```

```
local newProductor

function productor()
     local i = 0
     while true do
          i = i + 1
          send(i)     -- 将生产的物品发送给消费者
     end
end

function consumer()
     while true do
          local i = receive()     -- 从生产者那里得到物品
          print(i)
     end
end

function receive()
     local status, value = coroutine.resume(newProductor)
     return value
end

function send(x)
     coroutine.yield(x)     -- x表示需要发送的值，值返回以后，就挂起该协同程序
end

-- 启动程序
newProductor = coroutine.create(productor)
consumer()
```

以上实例执行输出结果为：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
……
```

Lua 文件 I/O

Lua I/O 库用于读取和处理文件。分为简单模式（和C一样）、完全模式。

- 简单模式（simple model）拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作。
- 完全模式（complete model） 使用外部的文件句柄来实现。它以一种面对对象的形式，将所有的文件操作定义为文件句柄的方法

简单模式在做一些简单的文件操作时较为合适。但是在进行一些高级的文件操作的时候，简单模式就显得力不从心。例如同时读取多个文件这样的操作，使用完全模式则较为合适。

打开文件操作语句如下：

```
file = io.open (filename [, mode])
```

mode 的值有：

模式	描述
r	以只读方式打开文件，该文件必须存在。
w	打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。
a	以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF符保留）
r+	以可读写方式打开文件，该文件必须存在。
w+	打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。
a+	与a类似，但此文件可读可写
b	二进制模式，如果文件是二进制文件，可以加上b
+	号表示对文件既可以读也可以写

简单模式

简单模式使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。

以下为 file.lua 文件代码，操作的文件为test.lua(如果没有你需要创建该文件)，代码如下：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 设置默认输入文件为 test.lua
io.input(file)

-- 输出文件第一行
print(io.read())

-- 关闭打开的文件
io.close(file)

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 设置默认输出文件为 test.lua
io.output(file)

-- 在文件最后一行添加 Lua 注释
io.write("--  test.lua 文件末尾注释")

-- 关闭打开的文件
io.close(file)
```

执行以上代码，你会发现，输出了 test.ua 文件的第一行信息，并在该文件最后一行添加了 lua 的注释。如我这边输出的是：

```
-- test.lua 文件
```

在以上实例中我们使用了 io."x" 方法，其中 io.read() 中我们没有带参数，参数可以是下表中的一个：

模式	描述
"*n"	读取一个数字并返回它。例：file.read("*n")
"*a"	从当前位置读取整个文件。例：file.read("*a")
"*l"（默认）	读取下一行，在文件尾 (EOF) 处返回 nil。例：file.read("*l")
number	返回一个指定字符个数的字符串，或在 EOF 时返回 nil。例：file.read(5)

其他的 io 方法有：

- **io.tmpfile():**返回一个临时文件句柄，该文件以更新模式打开，程序结束时自动删除
- **io.type(file):** 检测obj是否一个可用的文件句柄
- **io.flush():** 向文件写入缓冲中的所有数据
- **io.lines(optional file name):** 返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回nil,但不关闭文件

完全模式

通常我们需要在同一时间处理多个文件。我们需要使用 file:function_name 来代替 io.function_name 方法。以下实例演示了如同同时处理同一个文件:

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 输出文件第一行
print(file:read())

-- 关闭打开的文件
file:close()

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 在文件最后一行添加 Lua 注释
file:write("--test")

-- 关闭打开的文件
file:close()
```

执行以上代码，你会发现，输出了 test.ua 文件的第一行信息，并在该文件最后一行添加了 lua 的注释。如我这边输出的是：

```
-- test.lua 文件
```

read 的参数与简单模式一致。

其他方法:

- **file:seek(optional whence, optional offset):** 设置和获取当前文件位置,成功则返回最终的文件位置(按字节),失败则返回nil加错误信息。参数 whence 值可以是:
  - "set": 从文件头开始
  - "cur": 从当前位置开始[默认]
  - "end": 从文件尾开始
  - offset:默认为0不带参数file:seek()则返回当前位置,file:seek("set")则定位到文件头,file:seek("end")则定位到文件尾并返回文件大小
- **file:flush():** 向文件写入缓冲中的所有数据
- **io.lines(optional file name):** 打开指定的文件filename为读模式并返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回nil,并自动关闭文件。 若不带参数时io.lines() <=> io.input():lines(); 读取默认输入设备的内容，但结束时不关闭文件,如

  ```
  for line in io.lines("main.lua") do

      print(line)

      end
  ```

以下实例使用了 seek 方法，定位到文件倒数第 25 个位置并使用 read 方法的 *a 参数，即从当期位置(倒数第 25 个位置)读取整个文件。

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

file:seek("end",-25)
print(file:read("*a"))

-- 关闭打开的文件
file:close()
```

我这边输出的结果是：

```
st.lua 文件末尾--test
```

Lua 错误处理

程序运行中错误处理是必要的，在我们进行文件操作，数据转移及web service 调用过程中都会出现不可预期的错误。如果不注重错误信息的处理，就会照成信息泄露，程序无法运行等情况。

任何程序语言中，都需要错误处理。错误类型有：

- 语法错误
- 运行错误

语法错误

语法错误通常是由于对程序的组件（如运算符、表达式）使用不当引起的。一个简单的实例如下：

```
-- test.lua 文件
a == 2
```

以上代码执行结果为：

```
lua: test.lua:2: syntax error near '=='
```

正如你所看到的，以上出现了语法错误，一个 "=" 号跟两个 "=" 号是有区别的。一个 "=" 是赋值表达式两个 "=" 是比较运算。

另外一个实例:

```
for a= 1,10
   print(a)
end
```

执行以上程序会出现如下错误：

```
lua: test2.lua:2: 'do' expected near 'print'
```

语法错误比程序运行错误更简单，运行错误无法定位具体错误，而语法错误我们可以很快的解决，如以上实例我们只要在for语句下添加 do 即可：

```
for a= 1,10
do
   print(a)
end
```

运行错误

运行错误是程序可以正常执行，但是会输出报错信息。如下实例由于参数输入错误，程序执行时报错：

```
function add(a,b)
   return a+b
end

add(10)
```

当我们编译运行以下代码时，编译是可以成功的，但在运行的时候会产生如下错误：

```
lua: test2.lua:2: attempt to perform arithmetic on local 'b' (a nil value)
stack traceback:
    test2.lua:2: in function 'add'
    test2.lua:5: in main chunk
    [C]: ?
```

以下报错信息是由于程序缺少 b 参数引起的。

错误处理

我们可以使用两个函数：assert 和 error 来处理错误。实例如下：

```
local function add(a,b)
   assert(type(a) == "number", "a 不是一个数字")
   assert(type(b) == "number", "b 不是一个数字")
   return a+b
end
add(10)
```

执行以上程序会出现如下错误：

```
lua: test.lua:3: b 不是一个数字
stack traceback:
    [C]: in function 'assert'
    test.lua:3: in local 'add'
    test.lua:6: in main chunk
    [C]: in ?
```

实例中assert首先检查第一个参数，若没问题，assert不做任何事情；否则，assert以第二个参数作为错误信息抛出。

error函数

语法格式：

```
error (message [, level])
```

功能：终止正在执行的函数，并返回message的内容作为错误信息(error函数永远都不会返回)

通常情况下，error会附加一些错误位置的信息到message头部。

Level参数指示获得错误的位置:

- Level=1[默认]：为调用error位置(文件+行号)
- Level=2：指出哪个调用error的函数的函数
- Level=0:不添加错误位置信息

pcall 和 xpcall、debug

Lua中处理错误，可以使用函数pcall（protected call）来包装需要执行的代码。

pcall接收一个函数和要传递个后者的参数，并执行，执行结果：有错误、无错误；返回值true或者或false, errorinfo。

语法格式如下

```
if pcall(function_name, ….) then
-- 没有错误
else
-- 一些错误
end
```

简单实例：

```
> =pcall(function(i) print(i) end, 33)
33
true

> =pcall(function(i) print(i) error('error..') end, 33)
33
false        stdin:1: error..
```

```
> function f() return false,2 end
> if f() then print '1' else print '0' end
0
```

pcall以一种"保护模式"来调用第一个参数，因此pcall可以捕获函数执行中的任何错误。

通常在错误发生时，希望落得更多的调试信息，而不只是发生错误的位置。但pcall返回时，它已经销毁了调用栈的部分内容。

Lua提供了xpcall函数，xpcall接收第二个参数——一个错误处理函数，当错误发生时，Lua会在调用栈展看（unwind）前调用错误处理函数，于是就可以在这个函数中使用debug库来获取关于错误的额外信息了。

debug库提供了两个通用的错误处理函数:

- debug.debug：提供一个Lua提示符，让用户来价差错误的原因
- debug.traceback：根据调用栈来构建一个扩展的错误消息

> =xpcall(function(i) print(i) error('error..') end, function() print(debug.traceback()) end, 33) 33 stack traceback: stdin:1: in function 1>[C]: in function 'error' stdin:1: in function 1>[C]: in function 'xpcall' stdin:1: in main chunk [C]: in ? false nil
>
> 1>

1>

xpcall 使用实例 2:

```
function myfunction ()
   n = n/nil
end

function myerrorhandler( err )
   print( "ERROR:", err )
end

status = xpcall( myfunction, myerrorhandler )
print( status)
```

执行以上程序会出现如下错误：

```
ERROR:    test2.lua:2: attempt to perform arithmetic on global 'n' (a nil value)
false
```

Lua 调试(Debug)

Lua 提供了 debug 库用于提供创建我们自定义调速器的功能。Lua 本身并未有内置的调速器，但很多开发者共享了他们的 Lua 调速器代码。

Lua 中 debug 库包含以下函数：

sethook ([thread,] hook, mask [, count]):

方法	用途
debug():	进入一个用户交互模式，运行用户输入的每个字符串。 使用简单的命令以及其它调试设置，用户可以检阅全局变量和局部变量， 改变变量的值，计算一些表达式，等等。输入一行仅包含 cont 的字符串将结束这个函数， 这样调用者就可以继续向下运行。
getfenv(object):	返回对象的环境变量。
gethook(optional thread):	返回三个表示线程钩子设置的值： 当前钩子函数，当前钩子掩码，当前钩子计数
getinfo ([thread,] f [, what]):	返回关于一个函数信息的表。 你可以直接提供该函数， 也可以用一个数字 f 表示该函数。 数字 f 表示运行在指定线程的调用栈对应层次上的函数： 0 层表示当前函数（getinfo 自身）； 1 层表示调用 getinfo 的函数 （除非是尾调用，这种情况不计入栈）；等等。 如果 f 是一个比活动函数数量还大的数字， getinfo 返回 nil。
debug.getlocal ([thread,] f, local):	此函数返回在栈的 f 层处函数的索引为 local 的局部变量 的名字和值。 这个函数不仅用于访问显式定义的局部变量，也包括形参、临时变量等。
getmetatable(value):	把给定索引指向的值的元表压入堆栈。如果索引无效，或是这个值没有元表，函数将返回 0 并且不会向栈上压任何东西。
getregistry():	返回注册表表，这是一个预定义出来的表， 可以用来保存任何 C 代码想保存的 Lua 值。
getupvalue (f, up)	此函数返回函数 f 的第 up 个上值的名字和值。 如果该函数没有那个上值，返回 nil 。以 '(' （开括号）打头的变量名表示没有名字的变量 （去除了调试信息的代码块）。
	将一个函数作为钩子函数设入。 字符串 mask 以及数字 count 决定了钩子将在何时调用。 掩码是由下列字符组合成的字符串，每个字符有其含义： **'`c`':** 每当 Lua 调用一个函数时，调用钩子； **'`r`':** 每当 Lua 从一个函数内返回时，调用钩子； **'`l`':** 每当 Lua 进入新的一行时，调用钩子。
setlocal ([thread,] level, local, value):	这个函数将 value 赋给 栈上第 level 层函数的第 local 个局部变量。 如果没有那个变量，函数返回 nil 。 如果 level 越界，抛出一个错误。
setmetatable (value, table):	将 value 的元表设为 table （可以是 nil）。 返回 value。
setupvalue (f, up, value):	这个函数将 value 设为函数 f 的第 up 个上值。 如果函数没有那个上值，返回 nil 否则，返回该上值的名字。
traceback ([thread,] [message [, level]]):	如果 message 有，且不是字符串或 nil， 函数不做任何处理直接返回 message。 否则，它返回调用栈的栈回溯信息。 字符串可选项 message 被添加在栈回溯信息的开头。 数字可选项 level 指明从栈的哪一层开始回溯 （默认为 1 ，即调用 traceback 的那里）。

上表列出了我们常用的调试函数，接下来我们可以看些简单的例子：

```
function myfunction ()
print(debug.traceback("Stack trace"))
print(debug.getinfo(1))
print("Stack trace end")
    return 10
end
myfunction ()
print(debug.getinfo(1))
```

执行以上代码输出结果为：

```
Stack trace
stack traceback:
    test2.lua:2: in function 'myfunction'
    test2.lua:8: in main chunk
    [C]: ?
table: 0054C6C8
Stack trace end
```

在以实例中，我们使用到了 debug 库的 traceback 和 getinfo 函数， getinfo 函数用于返回函数信息的表。

另一个实例

我们经常需要调试函数的内的局部变量。我们可以使用 getupvalue 函数来设置这些局部变量。实例如下：

```
function newCounter ()
  local n = 0
  local k = 0
  return function ()
    k = n
    n = n + 1
    return n
    end
end

counter = newCounter ()
print(counter())
print(counter())

local i = 1

repeat
  name, val = debug.getupvalue(counter, i)
  if name then
    print ("index", i, name, "=", val)
    if(name == "n") then
        debug.setupvalue (counter,2,10)
    end
    i = i + 1
  end -- if
until not name

print(counter())
```

执行以上代码输出结果为：

```
1
2
index    1    k    =    1
index    2    n    =    2
11
```

在以上实例中，计数器在每次调用时都会自增1。实例中我们使用了 getupvalue 函数查看局部变量的当前状态。我们可以设置局部变量为新值。实例中，在设置前 n 的值为 2,使用 setupvalue 函数将其设置为 10。现在我们调用函数，执行后输出为 11 而不是 3。

调试类型

- 命令行调试
- 图形界面调试

命令行调试器有：RemDebug、clidebugger、ctrace、xdbLua、LuaInterface - Debugger、Rldb、ModDebug。

图形界调试器有：SciTE、Decoda、ZeroBrane Studio、akdebugger、luaedit。

Lua 垃圾回收

Lua 采用了自动内存管理。 这意味着你不用操心新创建的对象需要的内存如何分配出来， 也不用考虑在对象不再被使用后怎样释放它们所占用的内存。

Lua 运行了一个垃圾收集器来收集所有死对象 （即在 Lua 中不可能再访问到的对象）来完成自动内存管理的工作。 Lua 中所有用到的内存，如：字符串、表、用户数据、函数、线程、 内部结构等，都服从自动管理。

Lua 实现了一个增量标记-扫描收集器。 它使用这两个数字来控制垃圾收集循环： 垃圾收集器间歇率和垃圾收集器步进倍率。 这两个数字都使用百分数为单位 （例如：值 100 在内部表示 1 ）。

垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。 增大这个值会减少收集器的积极性。 当这个值比 100 小的时候，收集器在开启新的循环前不会有等待。 设置这个值为 200 就会让收集器等到总内存使用量达到 之前的两倍时才开始新的循环。

垃圾收集器步进倍率控制着收集器运作速度相对于内存分配速度的倍率。 增大这个值不仅会让收集器更加积极，还会增加每个增量步骤的长度。 不要把这个值设得小于 100 ， 那样的话收集器就工作的太慢了以至于永远都干不完一个循环。 默认值是 200 ，这表示收集器以内存分配的"两倍"速工作。

如果你把步进倍率设为一个非常大的数字 （比你的程序可能用到的字节数还大 10% ）， 收集器的行为就像一个 stop-the-world 收集器。 接着你若把间歇率设为 200 ， 收集器的行为就和过去的 Lua 版本一样了： 每次 Lua 使用的内存翻倍时，就做一次完整的收集。

垃圾回收器函数

Lua 提供了以下函数**collectgarbage ([opt [, arg]])**用来控制自动内存管理:

- **collectgarbage("collect"):** 做一次完整的垃圾收集循环。通过参数 opt 它提供了一组不同的功能：
- **collectgarbage("count"):** 以 K 字节数为单位返回 Lua 使用的总内存数。 这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。
- **collectgarbage("restart"):** 重启垃圾收集器的自动运行。
- **collectgarbage("setpause"):** 将 arg 设为收集器的 间歇率 （参见 §2.5）。 返回 间歇率 的前一个值。
- **collectgarbage("setstepmul"):** 返回 步进倍率 的前一个值。

- **collectgarbage("step"):** 单步运行垃圾收集器。 步长"大小"由 arg 控制。 传入 0 时，收集器步进（不可分割的）一步。 传入非 0 值， 收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。 如果收集器结束一个循环将返回 true 。
- **collectgarbage("stop"):** 停止垃圾收集器的运行。 在调用重启前，收集器只会因显式的调用运行。

以下演示了一个简单的垃圾回收实例:

```
mytable = {"apple", "orange", "banana"}

print(collectgarbage("count"))

mytable = nil

print(collectgarbage("count"))

print(collectgarbage("collect"))

print(collectgarbage("count"))
```

执行以上程序，输出结果如下(注意内存使用的变化)：

```
20.9560546875
20.9853515625
0
19.4111328125
```

Lua 面向对象

面向对象编程（Object Oriented Programming，OOP）是一种非常流行的计算机编程架构。

以下几种编程语言都支持面向对象编程：

- C++
- Java
- Objective-C
- Smalltalk
- C#
- Ruby

面向对象特征

- 1） 封装：指能够把一个实体的信息、功能、响应都装入一个单独的对象中的特性。
- 2） 继承：继承的方法允许在不改动原程序的基础上对其进行扩充，这样使得原功能得以保存，而新功能也得以扩展。这有利于减少重复编码，提高软件的开发效率。
- 3） 多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，可以通过指向基类的指针，来调用实现派生类中的方法。
- 4）抽象：抽象(Abstraction)是简化复杂的现实问题的途径，它可以为具体问题找到最恰当的类定义，并且可以在最恰当的继承级别解释问题。

Lua 中面向对象

我们知道，对象由属性和方法组成。LUA中最基本的结构是table，所以需要用table来描述对象的属性。

lua中的function可以用来表示方法。那么LUA中的类可以通过table + function模拟出来。

至于继承，可以通过metetable模拟出来（不推荐用，只模拟最基本的对象大部分时间够用了）。

Lua中的表不仅在某种意义上是一种对象。像对象一样，表也有状态（成员变量）；也有与对象的值独立的本性，特别是拥有两个不同值的对象（table）代表两个不同的对象；一个对象在不同的时候也可以有不同的值，但他始终是一个对象；与对象类似，表的生命周期与其由什么创建、在哪创建没有关系。对象有他们的成员函数，表也有：

```
Account = {balance = 0}
function Account.withdraw (v)
    Account.balance = Account.balance - v
end
```

这个定义创建了一个新的函数，并且保存在Account对象的withdraw域内，下面我们可以这样调用：

```
Account.withdraw(100.00)
```

一个简单实例

以下简单的类包含了三个属性： area, length 和 breadth，printArea方法用于打印计算结果：

```
-- Meta class
Rectangle = {area = 0, length = 0, breadth = 0}

-- 派生类的方法 new
function Rectangle:new (o,length,breadth)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  self.length = length or 0
  self.breadth = breadth or 0
  self.area = length*breadth;
  return o
end

-- 派生类的方法 printArea
function Rectangle:printArea ()
  print("矩形面积为 ",self.area)
end
```

创建对象

创建对象是位类的实例分配内存的过程。每个类都有属于自己的内存并共享公共数据。

```
r = Rectangle:new(nil,10,20)
```

访问属性

我们可以使用点号(.)来访问类的属性：

```
print(r.length)
```

访问成员函数

我们可以使用冒号(:)来访问类的属性：

```
r:printArea()
```

内存在对象初始化时分配。

完整实例

以下我们演示了 Lua 面向对象的完整实例：

```
-- Meta class
Shape = {area = 0}

-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end

-- 基础类方法 printArea
function Shape:printArea ()
  print("面积为 ",self.area)
end

-- 创建对象
myshape = Shape:new(nil,10)

myshape:printArea()
```

执行以上程序，输出结果为：

```
面积为     100
```

Lua 继承

继承是指一个对象直接使用另一对象的属性和方法。可用于扩展基础类的属性和方法。

以下演示了一个简单的继承实例：

```
 -- Meta class
Shape = {area = 0}
-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- 基础类方法 printArea
function Shape:printArea ()
  print("面积为 ",self.area)
end
```

接下来的实例，Square 对象继承了 Shape 类:

```
Square = Shape:new()
-- Derived class method new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
  self.__index = self
  return o
end
```

完整实例

以下实例我们继承了一个简单的类，来扩展派生类的方法，派生类中保留了继承类的成员变量和方法：

```
 -- Meta class
Shape = {area = 0}
-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- 基础类方法 printArea
function Shape:printArea ()
  print("面积为 ",self.area)
end

-- 创建对象
myshape = Shape:new(nil,10)
myshape:printArea()

Square = Shape:new()
-- 派生类方法 new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
  self.__index = self
  return o
end

-- 派生类方法 printArea
function Square:printArea ()
  print("正方形面积为 ",self.area)
end

-- 创建对象
mysquare = Square:new(nil,10)
mysquare:printArea()

Rectangle = Shape:new()
-- 派生类方法 new
function Rectangle:new (o,length,breadth)
  o = o or Shape:new(o)
  setmetatable(o, self)
  self.__index = self
  self.area = length * breadth
  return o
end

-- 派生类方法 printArea
function Rectangle:printArea ()
  print("矩形面积为 ",self.area)
end

-- 创建对象
myrectangle = Rectangle:new(nil,10,20)
myrectangle:printArea()
```

执行以上代码，输出结果为：

```
面积为 	100
正方形面积为 	100
矩形面积为 	200
```

函数重写

Lua 中我们可以重写基础类的函数，在派生类中定义自己的实现方式：

```
-- 派生类方法 printArea
function Square:printArea ()
  print("正方形面积 ",self.area)
end
```

Lua 数据库访问

本文主要为大家介绍 Lua 数据库的操作库：LuaSQL。他是开源的，支持的数据库有：ODBC, ADO, Oracle, MySQL, SQLite 和 PostgreSQL。

本文为大家介绍MySQL的数据库连接。

LuaSQL 可以使用 LuaRocks 来安装可以根据需要安装你需要的数据库驱动。

LuaRocks 安装方法：

```
$ wget http://luarocks.org/releases/luarocks-2.2.1.tar.gz
$ tar zxpf luarocks-2.2.1.tar.gz
$ cd luarocks-2.2.1
$ ./configure; sudo make bootstrap
$ sudo luarocks install luasocket
$ lua
Lua 5.3.0 Copyright (C) 1994-2015 Lua.org, PUC-Rio
> require "socket"
```

Window 下安装 LuaRocks：https://github.com/keplerproject/luarocks/wiki/Installation-instructions-for-Windows

安装不同数据库驱动：

```
luarocks install luasql-sqlite3
luarocks install luasql-postgres
luarocks install luasql-mysql
luarocks install luasql-sqlite
luarocks install luasql-odbc
```

你也可以使用源码安装方式，Lua Github 源码地址：https://github.com/keplerproject/luasql

Lua 连接MySql 数据库：

```
require "luasql.mysql"

--创建环境对象
env = luasql.mysql()

--连接数据库
conn = env:connect("数据库名","用户名","密码","IP地址",端口)

--设置数据库的编码格式
conn:execute"SET NAMES UTF8"

--执行数据库操作
cur = conn:execute("select * from role")

row = cur:fetch({},"a")

--文件对象的创建
file = io.open("role.txt","w+");

while row do
    var = string.format("%d %s\n", row.id, row.name)

    print(var)

    file:write(var)

    row = cur:fetch(row,"a")
end

file:close()  --关闭文件对象
conn:close()  --关闭数据库连接
env:close()   --关闭数据库环境
```

</stdin:1></stdin:1></p这里注意对返回值的逻辑判断：<>

W3School Scala 教程

作者：W3School

来源：Scala 教程

Scala 教程

Scala 是一门多范式（multi-paradigm）的编程语言，设计初衷是要集成面向对象编程和函数式编程的各种特性。

Scala 运行在Java虚拟机上，并兼容现有的Java程序。

Scala 源代码被编译成Java字节码，所以它可以运行于JVM之上，并可以调用现有的Java类库。

谁适合阅读本教程？

本教程适合想从零开始学习 Scala 编程语言的开发人员。当然本教程也会对一些模块进行深入，让你更好的了解 Scala 的应用。

学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过Java编程语言，将有助于你更快的了解 Scala 编程。

学习 Java 教程。

第一个 Scala 程序：Hello World

以下是用 Scala 编写的典型 Hello World 程序：

实例（HelloWorld.scala）

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, world!")
    }
}
```

运行实例？

将以上代码保存为 HelloWorld.scala 文件，执行以上 scala 程序（你也可以直接在线执行）：

```
$ scalac HelloWorld.scala
$ scala HelloWorld.scala
```

输出结果为：

```
Hello, world!
```

相关文档推荐

以下是一份 Scala语言规范.pdf 文档，可作为学习参考：

Scala 简介

Scala 是 Scalable Language 的简写，是一门多范式的编程语言

联邦理工学院洛桑（EPFL）的Martin Odersky于2001年基于Funnel的工作开始设计Scala。

Funnel是把函数式编程思想和Petri网相结合的一种编程语言。

Odersky先前的工作是Generic Java和javac（Sun Java编译器）。Java平台的Scala于2003年底/2004年初发布。.NET平台的Scala发布于2004年6月。该语言第二个版本，v2.0，发布于2006年3月。

截至2009年9月，最新版本是版本2.7.6 。Scala 2.8预计的特性包括重写的Scala类库（Scala collections library）、方法的命名参数和默认参数、包对象（package object），以及Continuation。

2009年4月，Twitter宣布他们已经把大部分后端程序从Ruby迁移到Scala，其余部分也打算要迁移。此外， Wattzon已经公开宣称，其整个平台都已经是基于Scala基础设施编写的。

Scala 特性

面向对象特性

Scala是一种纯面向对象的语言，每个值都是对象。对象的数据类型以及行为由类和特质描述。

类抽象机制的扩展有两种途径：一种途径是子类继承，另一种途径是灵活的混入机制。这两种途径能避免多重继承的种种问题。

函数式编程

Scala也是一种函数式语言，其函数也能当成值来使用。Scala提供了轻量级的语法用以定义匿名函数，支持高阶函数，允许嵌套多层函数，并支持柯里化。Scala的case class及其内置的模式匹配相当于函数式编程语言中常用的代数类型。

更进一步，程序员可以利用Scala的模式匹配，编写类似正则表达式的代码处理XML数据。

静态类型

Scala具备类型系统，通过编译时检查，保证代码的安全性和一致性。类型系统具体支持以下特性：

- 泛型类
- 协变和逆变
- 标注
- 类型参数的上下限约束
- 把类别和抽象类型作为对象成员
- 复合类型
- 引用自己时显式指定类型
- 视图
- 多态方法

扩展性

Scala的设计秉承一项事实，即在实践中，某个领域特定的应用程序开发往往需要特定于该领域的语言扩展。Scala提供了许多独特的语言机制，可以以库的形式轻易无缝添加新的语言结构：

- 任何方法可用作前缀或后缀操作符
- 可以根据预期类型自动构造闭包。

并发性

Scala使用Actor作为其并发模型，Actor是类似线程的实体，通过邮箱发收消息。Actor可以复用线程，因此可以在程序中可以使用数百万个Actor,而线程只能创建数千个。在2.10之后的版本中，使用Akka作为其默认Actor实现。

谁使用了 Scala

- 2009年4月，Twitter宣布他们已经把大部分后端程序从Ruby迁移到Scala，其余部分也打算要迁移。
- 此外，Wattzon已经公开宣称，其整个平台都已经是基于Scala基础设施编写的。
- 瑞银集团把Scala用于一般产品中。
- Coursera把Scala作为服务器语言使用。

Scala Web 框架

以下列出了两个目前比较流行的 Scala 的 Web应用框架：

- Lift 框架
- Play 框架

Scala 安装

Scala 语言可以运行在Window、Linux、Unix、 Mac OS X等系统上。

Scala是基于java之上，大量使用java的类库和变量，必须使用Scala之前必须先安装Java（>1.5版本）。

Mac OS X 和 Linux 上安装 Scala

第一步：Java 设置

确保你本地以及安装了 JDK 1.5 以上版本，并且设置了 JAVA_HOME 环境变量及 JDK 的bin目录。

我们可以使用以下命令查看是否安装了 Java：

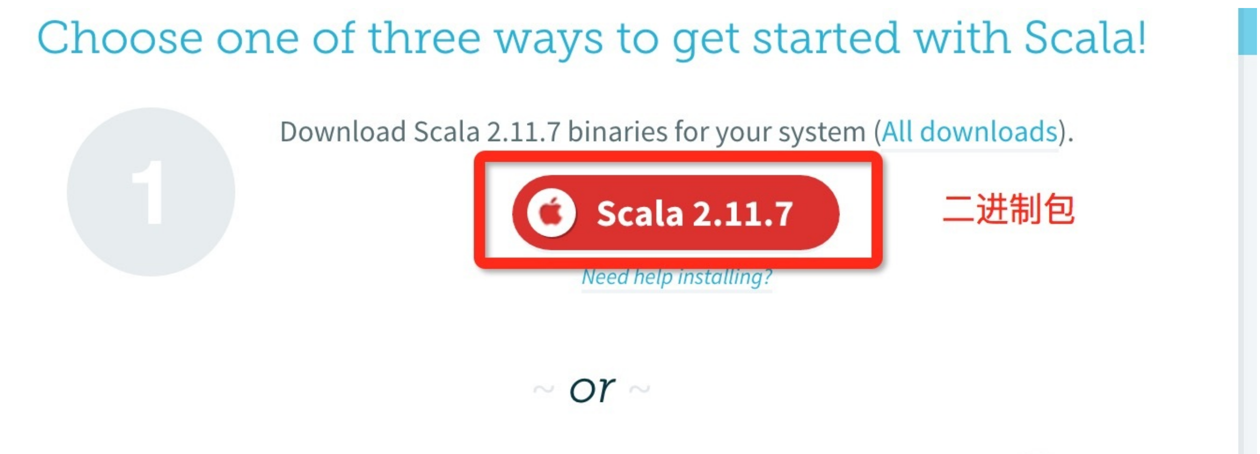
```
$ java -version
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
$
```

接着，我们可以查看是否安装了 Java 编译器。输入以下命令查看：

```
$ javac -version
javac 1.8.0_31 $
```

如果还为安装，可以参考我们的Java 开发环境配置。

接下来，我们可以从 Scala 官网地址 http://www.scala-lang.org/downloads 下载 Scala 二进制包，本教程我们将下载 *2.11.7*版本，如下图所示：



解压缩文件包，可将其移动至/usr/local/share下：

```
mv scala-2.11.7 scala                   # 重命名 Scala 目录
mv /download/scalapath /usr/local/share # 下载目录需要按你实际的下载路径
```

修改环境变量，如果不是管理员可使用 sudo 进入管理员权限，修改配置文件profile:

```
vim /etc/profile

或

sudo vim /etc/profile
```

在文件的末尾加入:

```
export PATH="$PATH:/usr/local/share/scala/bin"
```

:wq!保存退出，重启终端，执行 scala 命令，输出以下信息，表示安装成功：

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_31).
Type in expressions to have them evaluated.
Type :help for more information.
```

> 注意：在编译的时候，如果有中文会出现乱码现象，解决方法查看：Scala 中文乱码解决

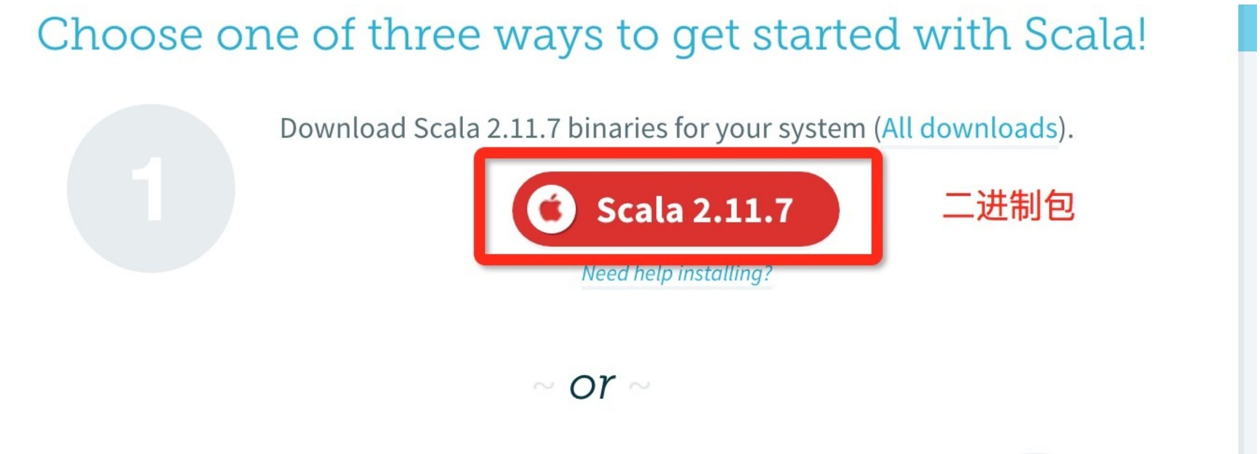
window 上安装 Scala

第一步：Java 设置

检测方法前文已说明，这里不再描述。

如果还为安装，可以参考我们的Java 开发环境配置。

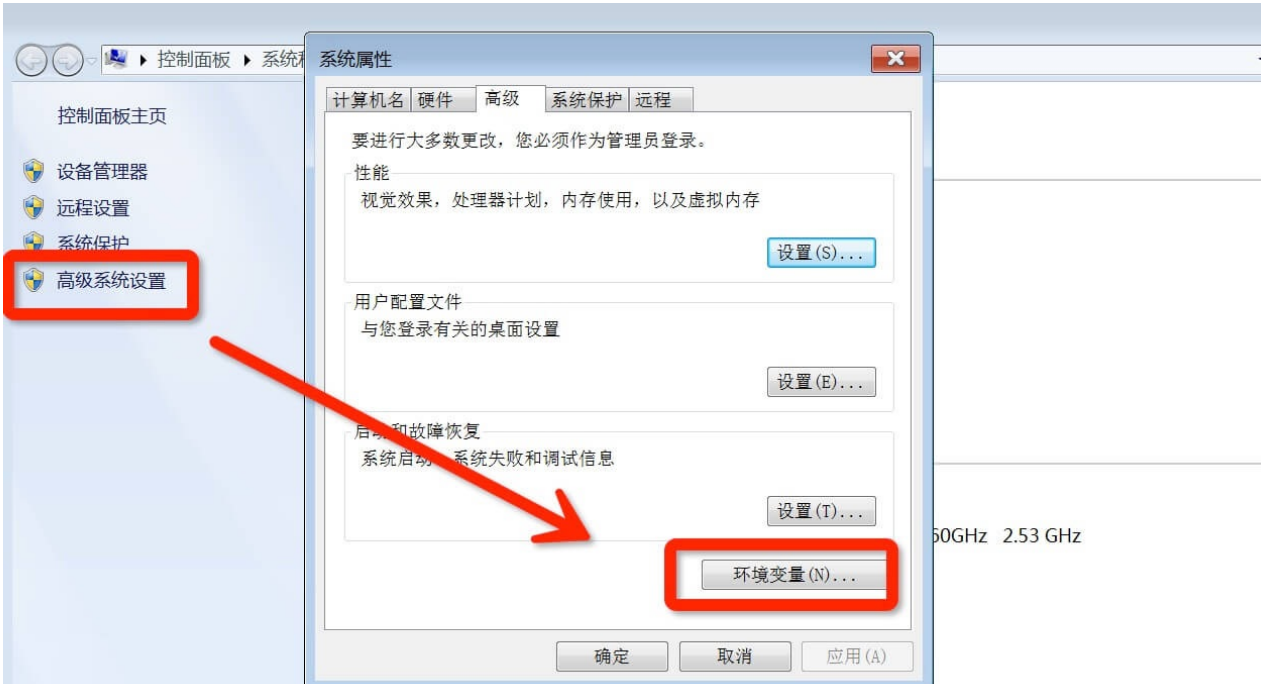
接下来，我们可以从 Scala 官网地址 http://www.scala-lang.org/downloads 下载 Scala 二进制包，本教程我们将下载 *2.11.7*版本，如下图所示：



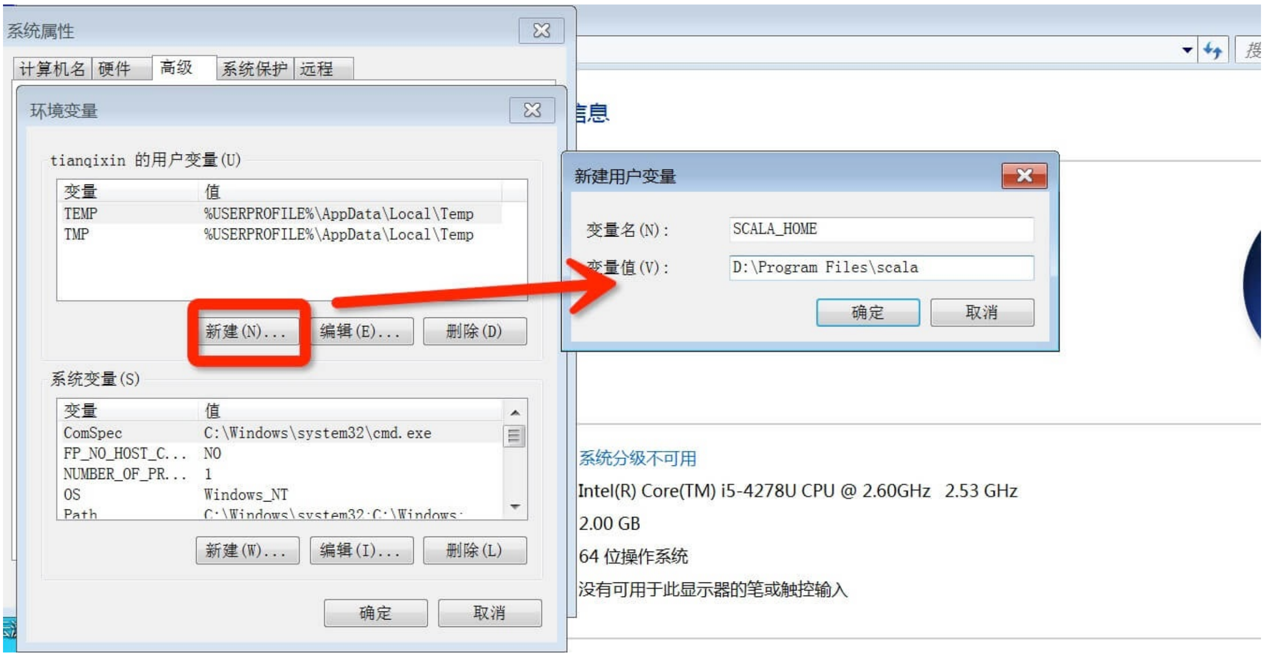
下载后，双击 msi 文件，一步步安装即可，安装过程你可以使用默认的安装目录。

安装好scala后，系统会自动提示，单击 finish，完成安装。

右击我的电脑，单击"属性"，进入如图所示页面。下面开始配置环境变量，右击【我的电脑】--【属性】--【高级系统设置】--【环境变量】，如图：

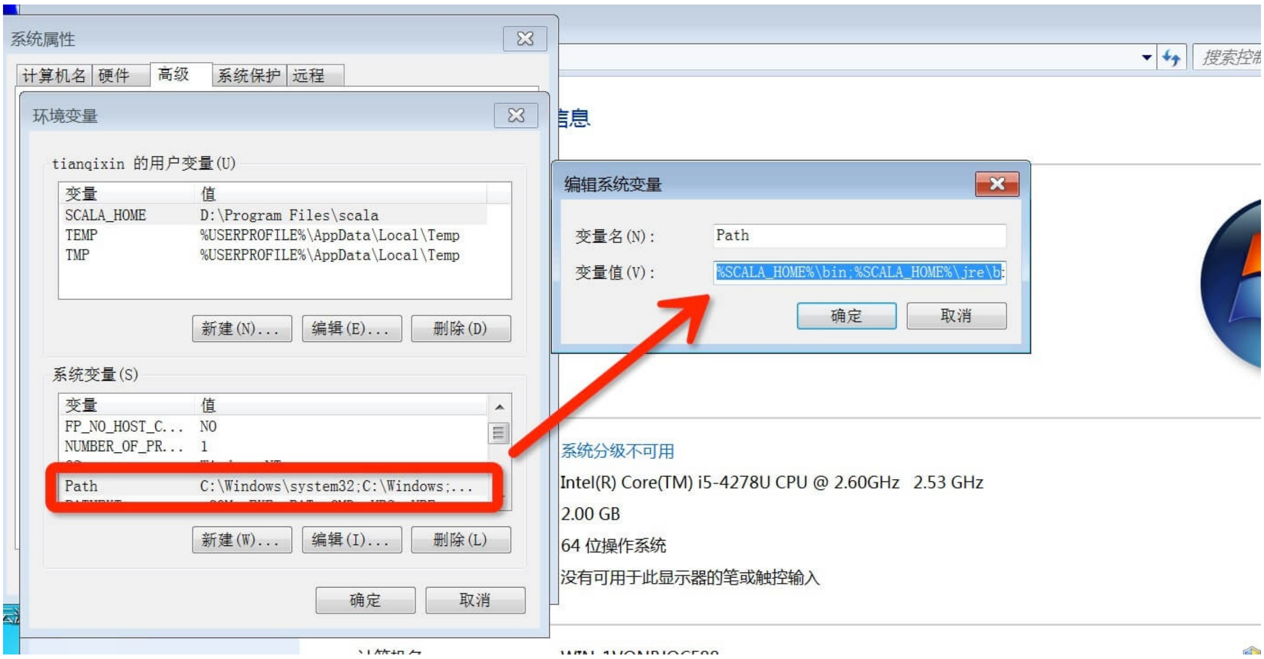


设置 SCALA_HOME 变量：单击新建，在变量名栏输入：**SCALA_HOME**: 变量值一栏输入：**D:\Program Files\scala** 也就是scala的安装目录，根据个人情况有所不同，如果安装在C盘，将"D"改成"C"即可。



设置 Path 变量：找到系统变量下的"Path"如图，单击编辑。在"变量值"一栏的最前面添加如下的路径： %SCALA_HOME%\bin;%SCALA_HOME%\jre\bin;

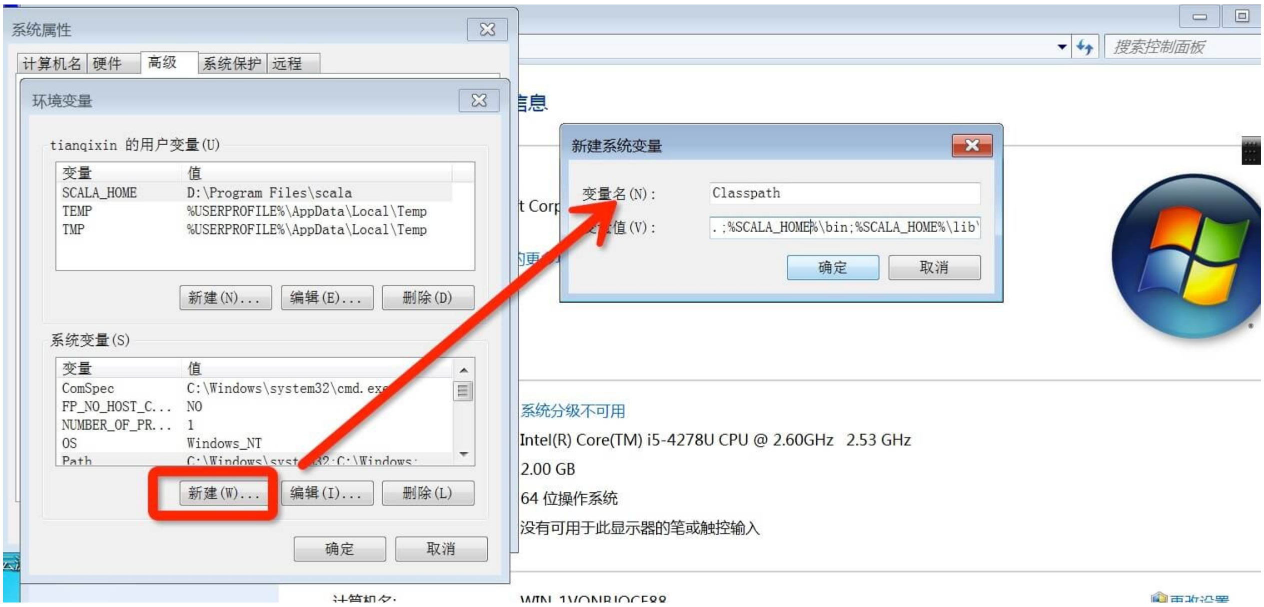
注意：后面的分号 ； 不要漏掉。



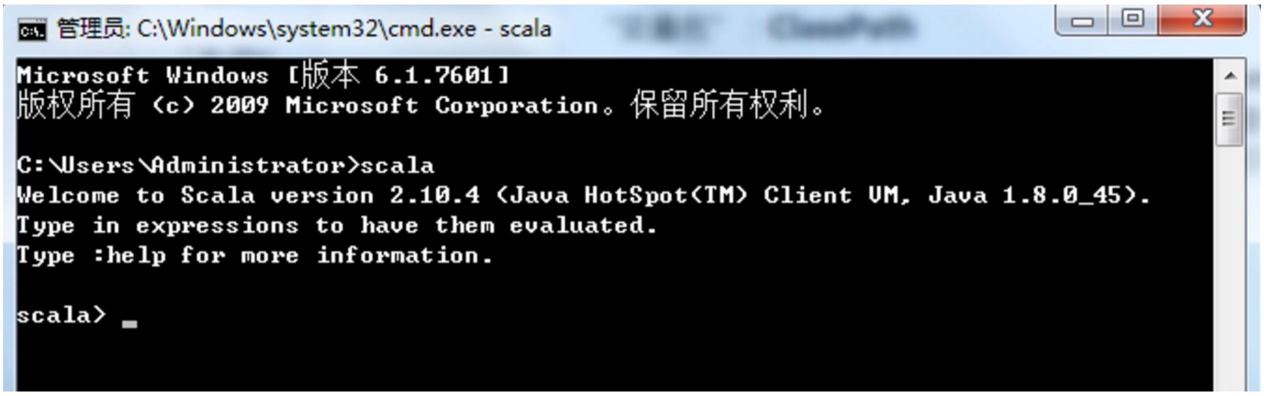
设置 Classpath 变量：找到找到系统变量下的"Classpath"如图，单击编辑，如没有，则单击"新建":

- "变量名"：ClassPath
- "变量值"：.;%SCALA_HOME%\bin;%SCALA_HOME%\lib\dt.jar;%SCALA_HOME%\lib\tools.jar.;

注意："变量值"最前面的 .; 不要漏掉。最后单击确定即可。



检查环境变量是否设置好了：调出"cmd"检查。单击 【开始】，在输入框中输入cmd，然后"回车"，输入 scala，然后回车，如环境变量设置ok，你应该能看到这些信息。



以下列出了不同系统放置的目录（可作为参考）：

系统环境	变量	值 (举例)
Unix	`$SCALA_HOME`	`/usr/local/share/scala`
	`$PATH`	`$PATH:$SCALA_HOME/bin`
Windows	`%SCALA_HOME%`	`c:\Progra~1\Scala`
	`%PATH%`	`%PATH%;%SCALA_HOME%\bin`

Scala 基础语法

如果你之前是一名 Java 程序员，并了解 Java 语言的基础知识，那么你能很快学会 Scala 的基础语法。

Scala 与 Java 的最大区别是：Scala 语句末尾的分号 ; 是可选的。

我们可以认为 Scala 程序是对象的集合，通过调用彼此的方法来实现消息传递。接下来我们来理解下，类，对象，方法，实例变量的概念：

- **对象 -** 对象有属性和行为。例如：一只狗的状属性有：颜色，名字，行为有：叫、跑、吃等。对象是一个类的实例。
- **类 -** 类是对象的抽象，而对象是类的具体实例。
- **方法 -** 方法描述的基本的行为，一个类可以包含多个方法。
- **字段 -** 每个对象都有它唯一的实例变量集合，即字段。对象的属性通过给字段赋值来创建。

第一个 Scala 程序

交互式编程

交互式编程不需要创建脚本文件，可以通过以下命令调用：

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_31).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 1 + 1
res0: Int = 2

scala> println("Hello World!")
Hello World!

scala>
```

脚本形式

我们也可以通过创建一个 HelloWorld.scala 的文件来执行代码，HelloWorld.scala 代码如下所示：

```
object HelloWorld {
   /* 这是我的第一个 Scala 程序
    * 以下程序将输出'Hello World!'
    */
   def main(args: Array[String]) {
      println("Hello, world!") // 输出 Hello World
   }
}
```

接下来我们使用 scalac 命令编译它：

```
$ scalac HelloWorld.scala
$ ls
HelloWorld$.class    HelloWorld.scala
HelloWorld.class
```

编译后我们可以看到目录下生成了 HelloWorld.class 文件，该文件可以在Java Virtual Machine (JVM)上运行。

编译后，我们可以使用以下命令来执行程序：

```
$ scala HelloWorld
Hello, world!
```

在线实例？

基本语法

Scala 基本语法需要注意以下几点：

- 区分大小写 - Scala是大小写敏感的，这意味着标识Hello 和 hello在Scala中会有不同的含义。
- 类名 - 对于所有的类名的第一个字母要大写。

  如果需要使用几个单词来构成一个类的名称，每个单词的第一个字母要大写。

  示例：*class MyFirstScalaClass*
- 方法名称 - 所有的方法名称的第一个字母用小写。

  如果若干单词被用于构成方法的名称，则每个单词的第一个字母应大写。

  示例：*def myMethodName()*
- 程序文件名 - 程序文件的名称应该与对象名称完全匹配。

  保存文件时，应该保存它使用的对象名称（记住Scala是区分大小写），并追加".scala"为文件扩展名。（如果文件名和对象名称不匹配，程序将无法编译）。

示例: 假设"HelloWorld"是对象的名称。那么该文件应保存为'HelloWorld.scala"

- **def main(args: Array[String])** - Scala程序从main()方法开始处理，这是每一个Scala程序的强制程序入口部分。

标识符

Scala 可以使用两种形式的标志符，字符数字和符号。

字符数字使用字母或是下划线开头，后面可以接字母或是数字，符号"$"在 Scala 中也看作为字母。然而以"$"开头的标识符为保留的 Scala 编译器产生的标志符使用，应用程序应该避免使用"$"开始的标识符，以免造成冲突。

Scala 的命名规则采用和 Java 类似的 camel 命名规则，首字符小写，比如 toString。类名的首字符还是使用大写。此外也应该避免使用以下划线结尾的标志符以避免冲突。符号标志符包含一个或多个符号，如+，:，? 等，比如:

```
+ ++ ::: < ?> :->
```

Scala 内部实现时会使用转义的标志符，比如:-> 使用 $colon$minus$greater 来表示这个符号。因此如果你需要在 Java 代码中访问:->方法，你需要使用 Scala 的内部名称 $colon$minus$greater。

混合标志符由字符数字标志符后面跟着一个或多个符号组成，比如 unary_+ 为 Scala 对+方法的内部实现时的名称。字面量标志符为使用"定义的字符串，比如 `x` `yield` 。

你可以在"之间使用任何有效的 Scala 标志符，Scala 将它们解释为一个 Scala 标志符，一个典型的使用为 Thread 的 yield 方法， 在 Scala 中你不能使用 Thread.yield()是因为 yield 为 Scala 中的关键字， 你必须使用 Thread.`yield`()来使用这个方法。

Scala 关键字

下表列出了 scala 保留关键字，我们不能使用以下关键字作为变量：

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	null
object	override	package	private
protected	return	sealed	super
this	throw	trait	try
true	type	val	var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Scala 注释

Scala 类似 Java 支持单行很多行注释。多行注释可以嵌套，但必须正确嵌套，一个注释开始符号对应一个结束符号。注释在 Scala 编译中会被忽略，实例如下：

```
object HelloWorld {
   /* 这是一个 Scala 程序
    * 这是一行注释
    * 这里演示了多行注释
    */
   def main(args: Array[String]) {
      // 输出 Hello World
      // 这是一个单行注释
      println("Hello, world!")
   }
}
```

空行和空格

一行中只有空格或者带有注释，Scala 会认为其是空行，会忽略它。标记可以被空格或者注释来分割。

换行符

Scala是面向行的语言，语句可以用分号（;）结束或换行符。Scala 程序里,语句末尾的分号通常是可选的。如果你愿意可以输入一个,但若一行里仅 有一个语句也可不写。 另一方面,如果一行里写多个语句那么分号是需要的。例如

```
val s = "菜鸟教程"; println(s)
```

Scala 包

定义包

Scala 使用 package 关键字定义包，在Scala将代码定义到某个包中有两种方式：

第一种方法和 Java 一样，在文件的头定义包名，这种方法就后续所有代码都放在该报中。比如：

```
package com.runoob
class HelloWorld
```

第二种方法有些类似 C#，如：

```
package com.runoob {
  class HelloWorld
}
```

第二种方法，可以在一个文件中定义多个包。

引用

Scala 使用 import 关键字引用包。

```
import java.awt.Color  // 引入Color

import java.awt._  // 引入包内所有成员

def handler(evt: event.ActionEvent) { // java.awt.event.ActionEvent
  ...  // 因为引入了java.awt，所以可以省去前面的部分
}
```

import语句可以出现在任何地方，而不是只能在文件顶部。import的效果从开始延伸到语句块的结束。这可以大幅减少名称冲突的可能性。

如果想要引入包中的几个成员，可以使用selector（选取器）：

```
import java.awt.{Color, Font}

// 重命名成员
import java.util.{HashMap => JavaHashMap}

// 隐藏成员
import java.util.{HashMap => _, _} // 引入了util包的所有成员，但是HashMap被隐藏了
```

> 注意：默认情况下，Scala 总会引入 java.lang._ 、 *scala._* 和 Predef._，这里也能解释，为什么以scala开头的包，在使用时都是省去scala.的。

Scala 数据类型

Scala 与 Java有着相同的数据类型，下表列出了 Scala 支持的数据类型：

数据类型	描述
Byte	8位有符号补码整数。数值区间为 -128 到 127
Short	16位有符号补码整数。数值区间为 -32768 到 32767
Int	32位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long	64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807
Float	32位IEEE754单精度浮点数
Double	64位IEEE754单精度浮点数
Char	16位无符号Unicode字符, 区间值为 U+0000 到 U+FFFF
String	字符序列
Boolean	true或false
Unit	表示无值，和其他语言中void等同。用作不返回任何结果的方法的结果类型。Unit只有一个实例值，写成()。
Null	null 或空引用
Nothing	Nothing类型在Scala的类层级的最低端；它是任何其他类型的子类型。
Any	Any是所有其他类的超类
AnyRef	AnyRef类是Scala里所有引用类(reference class)的基类

上表中列出的数据类型都是对象，也就是说scala没有java中的原生类型。在scala是可以对数字等基础类型调用方法的。

Scala 基础字面量

Scala 非常简单且直观。接下来我们会详细介绍 Scala 字面量。

整型字面量

整型字面量用于 Int 类型，如果表示 Long，可以在数字后面添加 L 或者小写 l 作为后缀。：

```
0
035
21
0xFFFFFFFF
0777L
```

浮点型字面量

如果浮点数后面有f或者F后缀时，表示这是一个Float类型，否则就是一个Double类型的。实例如下：

```
0.0
1e30f
3.14159f
1.0e100
.1
```

布尔型字面量

布尔型字面量有 true 和 false。

符号字面量

符号字面量被写成： **'<标识符>**， 这里 **<标识符>** 可以是任何字母或数字的标识（注意：不能以数字开头）。这种字面量被映射成预定义类scala.Symbol的实例。

如： 符号字面量 **'x** 是表达式 **scala.Symbol("x")** 的简写，符号字面量定义如下：

```
package scala
final case class Symbol private (name: String) {
   override def toString: String = "'" + name
}
```

字符字面量

在scala中字符类型表示为半角单引号(')中的字符，如下：

```
'a'
'\u0041'
'\n'
'\t'
```

其中 **** 表示转移字符，其后可以跟 **u0041** 数字或者 **\r\n** 等固定的转义字符。

字符串字面量

字符串表示方法是在双引号中(") 包含一系列字符，如：

```
"Hello,\nWorld!"
"菜鸟教程官网：www.runoob.com"
```

多行字符串的表示方法

多行字符串用三个双引号来表示分隔符，格式为：**""" ... """**。

实例如下：

```
val foo = """菜鸟教程
www.runoob.com
www.w3cschool.cc
www.runnoob.com
以上三个地址都能访问"""
```

Null 值

空值是 scala.Null 类型。

Scala.Null和scala.Nothing是用统一的方式处理Scala面向对象类型系统的某些"边界情况"的特殊类型。

Null类是null引用对象的类型，它是每个引用类（继承自AnyRef的类）的子类。Null不兼容值类型。

Scala 转义字符

下表列出了常见的转义字符：

转义字符	Unicode	描述
\b	\u0008	退格(BS) ，将当前位置移到前一列
\t	\u0009	水平制表(HT) （跳到下一个TAB位置）
\n	\u000c	换行(LF) ，将当前位置移到下一行开头
\f	\u000c	换页(FF)，将当前位置移到下页开头
\r	\u000d	回车(CR) ，将当前位置移到本行开头
\"	\u0022	代表一个双引号(")字符
\'	\u0027	代表一个单引号（'）字符
\\	\u005c	代表一个反斜线字符 '\'

0 到 255 间的 Unicode 字符可以用一个八进制转义序列来表示，即反斜线?\?后跟 最多三个八进制。

在字符或字符串中，反斜线和后面的字符序列不能构成一个合法的转义序列将会导致 编译错误。

以下实例演示了一些转义字符的使用：

```
object Test {
   def main(args: Array[String]) {
      println("Hello\tWorld\n\n" );
   }
}
```

运行实例？

执行以上代码输出结果如下所示：

```
$ scalac Test.scala
$ scala Test
Hello    World

$
```

Scala 变量

变量是一种使用方便的占位符，用于引用计算机内存地址，变量创建后会占用一定的内存空间。

基于变量的数据类型，操作系统会进行内存分配并且决定什么将被储存在保留内存中。因此，通过给变量分配不同的数据类型，你可以在这些变量中存储整数，小数或者字字母。

变量声明

在学习如何声明变量与常量之前，我们先来了解一些变量与常量。

- 一、变量： 在程序运行过程中其值可能发生改变的量叫做变量。如：时间，年龄。
- 二、常量 在程序运行过程中其值不会发生变化的量叫做常量。如：数值 3，字符'A'。

在 Scala 中，使用关键词 **"var"** 声明变量，使用关键词 **"val"** 声明常量。

声明变量实例如下：

```
var myVar : String = "Foo"
var myVar : String = "Too"
```

以上定义了变量 myVar，我们可以修改它。

声明常量实例如下：

```
val myVal :  String  =  "Foo"
```

以上定义了常量 myVal，它是不能修改的。如果程序尝试修改常量 myVal 的值，程序将会在编译时报错。

变量类型声明

变量的类型在变量名之后等号之前声明。定义变量的类型的语法格式如下：

```
var VariableName : DataType [=  Initial Value]

或

val VariableName : DataType [=  Initial Value]
```

变量声明不一定需要初始值，以下也是正确的：

```
var myVar :Int;
val myVal :String;
```

变量类型引用

在 Scala 中声明变量和常量不一定要指明数据类型，在没有指明数据类型的情况下，其数据类型是通过变量或常量的初始值推断出来的。

所以，如果在没有指明数据类型的情况下声明变量或常量必须要给出其初始值，否则将会报错。

```
var myVar = 10;
val myVal = "Hello, Scala!";
```

以上实例中，myVar 会被推断为 Int 类型，myVal 会被推断为 String 类型。

Scala 多个变量声明

Scala 支持多个变量的声明：

```
val xmax, ymax =  100  // xmax, ymax都声明为100
```

如果方法返回值是元组，我们可以使用 val 来声明一个元组：

```
val (myVar1:  Int, myVar2:  String)  =  Pair(40,  "Foo")
```

也可以不指定数据类型：

```
val (myVar1, myVar2)  =  Pair(40,  "Foo")
```

Scala 访问修饰符

Scala 访问修饰符基本和Java的一样，分别有：private，protected，public。

如果没有指定访问修饰符符，默认情况下，Scala对象的访问级别都是 public。

Scala 中的 private 限定符，比 Java 更严格，在嵌套类情况下，外层类甚至不能访问被嵌套类的私有成员。

私有(Private)成员

用private关键字修饰，带有此标记的成员仅在包含了成员定义的类或对象内部可见，同样的规则还适用内部类。

```
class Outer{
    class Inner{
    private def f(){println("f")}
    class InnerMost{
        f() // 正确
        }
    }
    (new Inner).f() //错误
}
```

(new Inner).f( ) 访问不合法是因为 f 在 Inner 中被声明为 private，而访问不在类Inner之内。

但在 InnerMost 里访问f就没有问题的，因为这个访问包含在 Inner 类之内。

Java中允许这两种访问，因为它允许外部类访问内部类的私有成员。

保护(Protected)成员

在 scala 中，对保护（Protected）成员的访问比 java 更严格一些。因为它只允许保护成员在定义了该成员的的类的子类中被访问。而在java中，用protected关键字修饰的成员，除了定义了该成员的类的子类可以访问，同一个包里的其他类也可以进行访问。

```
package p{
class Super{
    protected def f() {println("f")}
    }
    class Sub extends Super{
        f()
    }
    class Other{
        (new Super).f() //错误
    }
}
```

上例中，Sub 类对 f 的访问没有问题，因为 f 在 Super 中被声明为 protected，而 Sub 是 Super 的子类。相反，Other 对 f 的访问不被允许，因为 other 没有继承自 Super。而后者在 java 里同样被认可，因为 Other 与 Sub 在同一包里。

公共(Public)成员

Scala中，如果没有指定任何的修饰符，则默认为 public。这样的成员在任何地方都可以被访问。

```
class Outer {
   class Inner {
      def f() { println("f") }
      class InnerMost {
         f() // 正确
      }
   }
   (new Inner).f() // 正确因为 f() 是 public
}
```

作用域保护

Scala中，访问修饰符可以通过使用限定词强调。格式为:

```
private[x]

或

protected[x]
```

这里的x指代某个所属的包、类或单例对象。如果写成private[x],读作"这个成员除了对[…]中的类或[…]中的包中的类及它们的伴生对像可见外，对其它所有类都是private。

这种技巧在横跨了若干包的大型项目中非常有用，它允许你定义一些在你项目的若干子包中可见但对于项目外部的客户却始终不可见的东西。

```
package bobsrocckets{
    package navigation{
        private[bobsrockets] class Navigator{
         protected[navigation] def useStarChart(){}
         class LegOfJourney{
             private[Navigator] val distance = 100
             }
            private[this] var speed = 200
            }
        }
        package launch{
        import navigation._
        object Vehicle{
        private[launch] val guide = new Navigator
        }
    }
}
```

上述例子中，类Navigator被标记为private[bobsrockets]就是说这个类对包含在bobsrockets包里的所有的类和对象可见。

比如说，从Vehicle对象里对Navigator的访问是被允许的，因为对象Vehicle包含在包launch中，而launch包在bobsrockets中，相反，所有在包bobsrockets之外的代码都不能访问类Navigator。

Scala 运算符

一个运算符是一个符号，用于告诉编译器来执行指定的数学运算和逻辑运算。

Scala 含有丰富的内置运算符，包括以下几种类型：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符

接下来我们将为大家详细介绍以上各种运算符的应用。

算术运算符

下表列出了 Scala 支持的算术运算符。

假定变量 A 为 10，B 为 20：

运算符	描述	实例
+	加号	A + B 运算结果为 30
-	减号	A - B 运算结果为 -10
*	乘号	A * B 运算结果为 200
/	除号	B / A 运算结果为 2
%	取余	B % A 运算结果为 0

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      var c = 25;
      var d = 25;
      println("a + b = " + (a + b) );
      println("a - b = " + (a - b) );
      println("a * b = " + (a * b) );
      println("b / a = " + (b / a) );
      println("b % a = " + (b % a) );
      println("c % a = " + (c % a) );

   }
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
```

关系运算符

下表列出了 Scala 支持的关系运算符。

假定变量 A 为 10，B 为 20：

运算符	描述	实例
==	等于	(A == B) 运算结果为 false
!=	不等于	(A != B) 运算结果为 true
>	大于	(A > B) 运算结果为 false
<	小于	(A < B) 运算结果为 true
>=	大于等于	(A >= B) 运算结果为 false
<=	小于等于	(A <= B) 运算结果为 true

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      println("a == b = " + (a == b) );
      println("a != b = " + (a != b) );
      println("a > b = " + (a > b) );
      println("a < b = " + (a < b) );
      println("b >= a = " + (b >= a) );
      println("b <= a = " + (b <= a) );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

逻辑运算符

下表列出了 Scala 支持的逻辑运算符。

假定变量 A 为 1，B 为 0：

运算符	描述	实例
&&	逻辑与	(A && B) 运算结果为 false
||	逻辑或	(A || B) 运算结果为 true
!	逻辑非	!(A && B) 运算结果为 true

实例

```
object Test {
   def main(args: Array[String]) {
      var a = true;
      var b = false;

      println("a && b = " + (a&&b) );

      println("a || b = " + (a||b) );

      println("!(a && b) = " + !(a && b) );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a && b = false
a || b = true
!(a && b) = true
```

位运算符

位运算符用来对二进制位进行操作，~,&,|,^分别为取反，按位与与，按位与或，按位与异或运算，如下表实例：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

如果指定 A = 60; 及 B = 13; 两个变量对应的二进制为：

```
A = 0011 1100

B = 0000 1101

-------位运算----------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

Scala 中的按位运算法则如下：

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 ， 二进制解释 ： 0000 1100
|	按位或运算符	(a | b) 输出结果 61 ， 二进制解释 ： 0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49 ， 二进制解释 ： 0011 0001
~	按位取反运算符	(~a) 输出结果 -61 ， 二进制解释 ： 1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240 ， 二进制解释 ： 1111 0000
>>	右移动运算符	a >> 2 输出结果 15 ， 二进制解释 ： 0000 1111
>>>	无符号右移	A >>>2 输出结果 15, 二进制解释: 0000 1111

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 60;           /* 60 = 0011 1100 */
      var b = 13;           /* 13 = 0000 1101 */
      var c = 0;

      c = a & b;            /* 12 = 0000 1100 */
      println("a & b = " + c );

      c = a | b;            /* 61 = 0011 1101 */
      println("a | b = " + c );

      c = a ^ b;            /* 49 = 0011 0001 */
      println("a ^ b = " + c );

      c = ~a;               /* -61 = 1100 0011 */
      println("~a = " + c );

      c = a << 2;           /* 240 = 1111 0000 */
      println("a << 2 = " + c );

      c = a >> 2;           /* 215 = 1111 */
      println("a >> 2  = " + c );

      c = a >>> 2;          /* 215 = 0000 1111 */
      println("a >>> 2 = " + c );
   }
}
```

执行以上代码，输出结果为 ：

```
$ scalac Test.scala
$ scala Test
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 2  = 15
a >>> 2 = 15
```

赋值运算符

以下列出了 Scala 语言支持的赋值运算符:

运算符	描述	实例
=	简单的赋值运算，指定右边操作数赋值给左边的操作数。	C = A + B 将 A + B 的运算结果赋值给 C
+=	相加后再赋值，将左右两边的操作数相加后再赋值给左边的操作数。	C += A 相当于 C = C + A
-=	相减后再赋值，将左右两边的操作数相减后再赋值给左边的操作数。	C -= A 相当于 C = C - A
*=	相乘后再赋值，将左右两边的操作数相乘后再赋值给左边的操作数。	C = *A* 相当于 *C* = *C* A
/=	相除后再赋值，将左右两边的操作数相除后再赋值给左边的操作数。	C /= A 相当于 C = C / A
%=	求余后再赋值，将左右两边的操作数求余后再赋值给左边的操作数。	C %= A is equivalent to C = C % A
<<=	按位左移后再赋值	C <<= 2 相当于 C = C << 2
>>=	按位右移后再赋值	C >>= 2 相当于 C = C >> 2
&=	按位与运算后赋值	C &= 2 相当于 C = C & 2
^=	按位异或运算符后再赋值	C ^= 2 相当于 C = C ^ 2
|=	按位或运算后再赋值	C |= 2 相当于 C = C | 2

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      var c = 0;

      c = a + b;
      println("c = a + b  = " + c );

      c += a ;
      println("c += a  = " + c );

      c -= a ;
      println("c -= a = " + c );

      c *= a ;
      println("c *= a = " + c );

      a = 10;
      c = 15;
      c /= a ;
      println("c /= a  = " + c );

      a = 10;
      c = 15;
      c %= a ;
      println("c %= a  = " + c );

      c <<= 2 ;
      println("c <<= 2  = " + c );

      c >>= 2 ;
      println("c >>= 2  = " + c );

      c >>= 2 ;
      println("c >>= a  = " + c );

      c &= a ;
      println("c &= 2  = " + c );

      c ^= a ;
      println("c ^= a  = " + c );

      c |= a ;
      println("c |= a  = " + c );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
c = a + b  = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a  = 1
c %= a  = 5
c <<= 2  = 20
c >>= 2  = 5
c >>= a  = 1
c &= 2  = 0
c ^= a  = 10
c |= a  = 10
```

运算符优先级

在一个表达式中可能包含多个有不同运算符连接起来的、具有不同数据类型的数据对象；由于表达式有多种运算，不同的运算顺序可能得出不同结果甚至出现错误运算错误，因为当表达式中含多种运算时，必须按一定顺序进行结合，才能保证运算的合理性和结果的正确性、唯一性。

优先级从上到下依次递减，最上面具有最高的优先级，逗号操作符具有最低的优先级。

相同优先级中，按结合顺序计算。大多数运算是从左至右计算，只有三个优先级是从右至左结合的，它们是单目运算符、条件运算符、赋值运算符。

基本的优先级需要记住：

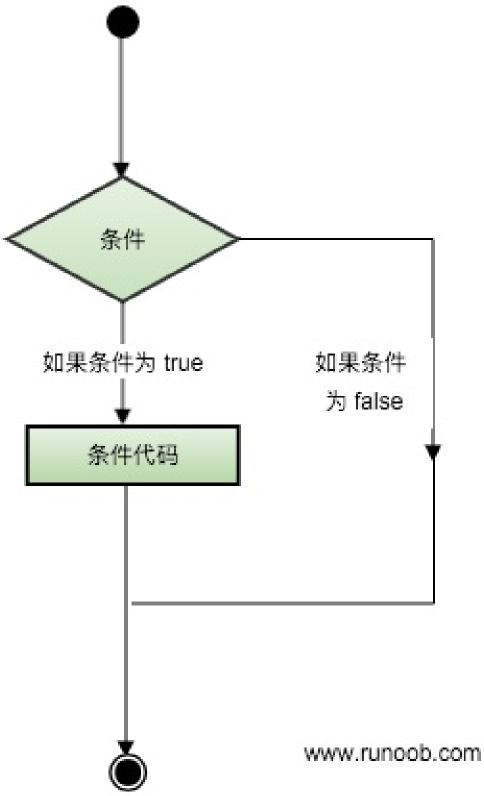
- 指针最优，单目运算优于双目运算。如正负号。
- 先乘除（模），后加减。
- 先算术运算，后移位运算，最后位运算。请特别注意：1 << 3 + 2 & 7 等价于 (1 << (3 + 2))&7
- 逻辑运算最后计算

运算符类型	运算符	结合方向
表达式运算	() [] . expr++ expr--	左到右
一元运算符	& + - ! ~ ++expr --expr * / % + - >> << > < >= <= == !=	右到左
位运算符	& ^ | && ||	左到右
三元运算符	?:	右到左
赋值运算符	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

Scala IF...ELSE 语句

Scala IF...ELSE 语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程:



if 语句

if 语句有布尔表达式及之后的语句块组成。

语法

if 语句的语法格式如下：

```
if(布尔表达式)
{
   // 如果布尔表达式为 true 则执行该语句块
}
```

如果布尔表达式为 true 则执行大括号内的语句块，否则跳过大括号内的语句块，执行大括号之后的语句块。

实例

```
object Test {
   def main(args: Array[String]) {
      var x = 10;

      if( x < 20 ){
         println("x < 20");
      }
   }
}
```

运行实例 ?

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x < 20
```

if...else 语句

if 语句后可以紧跟 else 语句，else 内的语句块可以在布尔表达式为 false 的时候执行。

语法

if...else 的语法格式如下：

```
if(布尔表达式){
   // 如果布尔表达式为 true 则执行该语句块
}else{
   // 如果布尔表达式为 false 则执行该语句块
}
```

实例

```
object Test {
   def main(args: Array[String]) {
      var x = 30;

      if( x < 20 ){
         println("x 小于 20");
      }else{
         println("x 大于 20");
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 大于 20
```

if...else if...else 语句

if 语句后可以紧跟 else if...else 语句，在多个条件判断语句的情况下很有用。

语法

if...else if...else 语法格式如下：

```
if(布尔表达式 1){
   // 如果布尔表达式 1 为 true 则执行该语句块
}else if(布尔表达式 2){
   // 如果布尔表达式 2 为 true 则执行该语句块
}else if(布尔表达式 3){
   // 如果布尔表达式 3 为 true 则执行该语句块
}else {
   // 如果以上条件都为 false 执行该语句块
}
```

实例

```
object Test {
   def main(args: Array[String]) {
      var x = 30;

      if( x == 10 ){
         println("X 的值为 10");
      }else if( x == 20 ){
         println("X 的值为 20");
      }else if( x == 30 ){
         println("X 的值为 30");
      }else{
         println("无法判断 X 的值");
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
X 的值为 30
```

if...else 嵌套语句

if...else 嵌套语句可以实现在 if 语句内嵌入一个或多个 if 语句。

语法

if...else 嵌套语句语法格式如下：

```
if(布尔表达式 1){
   // 如果布尔表达式 1 为 true 则执行该语句块
   if(布尔表达式 2){
      // 如果布尔表达式 2 为 true 则执行该语句块
   }
}
```

else if...else 的嵌套语句 类似 if...else 嵌套语句。

实例

```
object Test {
   def main(args: Array[String]) {
        var x = 30;
        var y = 10;

         if( x == 30 ){
            if( y == 10 ){
            println("X = 30 , Y = 10");
         }
      }
   }
}
```

执行以上代码，输出结果为：

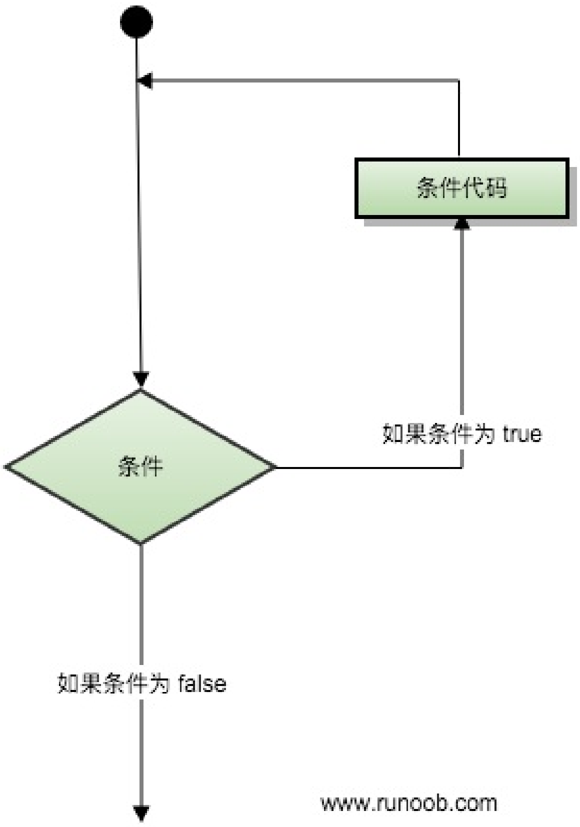
```
$ scalac Test.scala
$ scala Test
X =  30  , Y =  10
```

Scala 循环

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了更为复杂执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的流程图：



循环类型

Scala 语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	运行一系列语句，如果条件为true，会重复运行，直到条件变为false。
do...while 循环	类似 while 语句区别在于判断循环条件之前，先执行一次循环的代码块。
for 循环	用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。

循环控制语句

循环控制语句改变你代码的执行顺序，通过它你可以实现代码的跳转。Scala 以下几种循环控制语句：

Scala 不支持 break 或 continue 语句，但从 2.8 版本后提供了一种中断循环的方式，点击以下链接查看详情。

控制语句	描述
break 语句	中断循环

无限循环

如果条件永远为 true，则循环将变成无限循环。我们可以使用 while 语句来实现无限循环：

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      // 无限循环
      while( true ){
         println( "a 的值为 : " + a );
      }
   }
}
```

以上代码执行后循环会永久执行下去，你可以使用 Ctrl + C 键来中断无限循环。

Scala while 循环

只要给定的条件为 true，Scala 语言中的 **while** 循环语句会重复执行循环体内的代码块。

语法

Scala 语言中 **while** 循环的语法：

```
while(condition)
{
   statement(s);
}
```

在这里，**statement(s)** 可以是一个单独的语句，也可以是几个语句组成的代码块。**condition** 可以是任意的表达式，当为任意非零值时都为 true。当条件为 true 时执行循环。

当条件为 false 时，程序流将继续执行紧接着循环的下一条语句。

流程图

在这里，*while* 循环的关键点是循环可能一次都不会执行。当条件为 false 时，会跳过循环主体，直接执行紧接着 while 循环的下一条语句。

实例

```
object Test {
   def main(args: Array[String]) {
      // 局部变量
      var a = 10;

      // while 循环执行
      while( a < 20 ){
         println( "Value of a: " + a );
         a = a + 1;
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Scala do...while 循环

不像 while 循环在循环头部测试循环条件, Scala 语言中，do...while 循环是在循环的尾部检查它的条件。

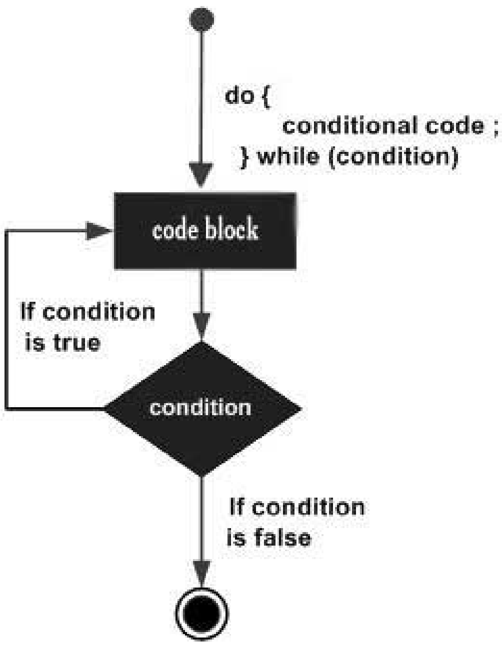
do...while 循环与 while 循环类似，但是 do...while 循环会确保至少执行一次循环。

语法

Scala 语言中 **while** 循环的语法：

```
do {
   statement(s);
} while( condition );
```

流程图



请注意，条件表达式出现在循环的尾部，所以循环中的 statement(s) 会在条件被测试之前至少执行一次。

如果条件为 true，控制流会跳转回上面的 do，然后重新执行循环中的 statement(s)。

这个过程会不断重复，直到给定条件变为 false 为止。

实例

```
object Test {
   def main(args: Array[String]) {
      // 局部变量
      var a = 10;

      // do 循环
      do{
         println( "Value of a: " + a );
         a = a + 1;
      }while( a < 20 )
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Scala do...while 循环

for 循环允许您编写一个执行指定次数的循环控制结构。

语法

Scala 语言中 **for** 循环的语法：

```
for( var x <- Range ){
   statement(s);
}
```

以上语法中，**Range** 可以是一个数字区间表示 **i to j** ，或者 **i until j**。左箭头 **<-** 用于为变量 x 赋值。

实例

以下是一个使用了 **i to j** 语法(包含 j)的实例:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      // for 循环
      for( a <- 1 to 10){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 10
```

以下是一个使用了 **i until j** 语法(不包含 j)的实例:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      // for 循环
      for( a <- 1 until 10){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
```

在 **for** 循环 中你可以使用分号 (;) 来设置多个区间，它将迭代给定区间所有的可能值。以下实例演示了两个区间的循环实例：

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      var b = 0;
      // for 循环
      for( a <- 1 to 3; b <- 1 to 3){
         println( "Value of a: " + a );
         println( "Value of b: " + b );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a: 1
Value of b: 1
Value of a: 1
Value of b: 2
Value of a: 1
Value of b: 3
Value of a: 2
Value of b: 1
Value of a: 2
Value of b: 2
Value of a: 2
Value of b: 3
Value of a: 3
Value of b: 1
Value of a: 3
Value of b: 2
Value of a: 3
Value of b: 3
```

for 循环集合

for 循环集合的语法如下：

```
for( var x <- List ){
   statement(s);
}
```

以上语法中， **List** 变量是一个集合，for 循环会迭代所有集合的元素。

实例

以下实例将循环数字集合。我们使用 *List()* 来创建集合。再以后章节我们会详细介绍集合。

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6);

      // for 循环
      for( a <- numList ){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
```

for 循环过滤

Scala 可以使用一个或多个 **if** 语句来过滤一些元素。

以下是在 for 循环中使用过滤器的语法。

```
for( var x <- List
      if condition1; if condition2...
   ){
   statement(s);
```

你可以使用分号(;)来为表达式添加一个或多个的过滤条件。

实例

以下是 for 循环中过滤的实例：

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      // for 循环
      for( a <- numList
           if a != 3; if a < 8 ){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 4
value of a: 5
value of a: 6
value of a: 7
```

for 使用 yield

你可以将 for 循环的返回值作为一个变量存储。语法格式如下：

```
var retVal = for{ var x <- List
     if condition1; if condition2...
}yield x
```

注意大括号中用于保存变量和条件，*retVal* 是变量， 循环中的 yield 会把当前的元素记下来，保存在集合中，循环结束后将返回该集合。

实例

以下实例演示了 for 循环中使用 yield：

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      // for 循环
      var retVal = for{ a <- numList
                        if a != 3; if a < 8
                      }yield a

      // 输出返回值
      for( a <- retVal){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 4
value of a: 5
value of a: 6
value of a: 7
```

Scala break 语句

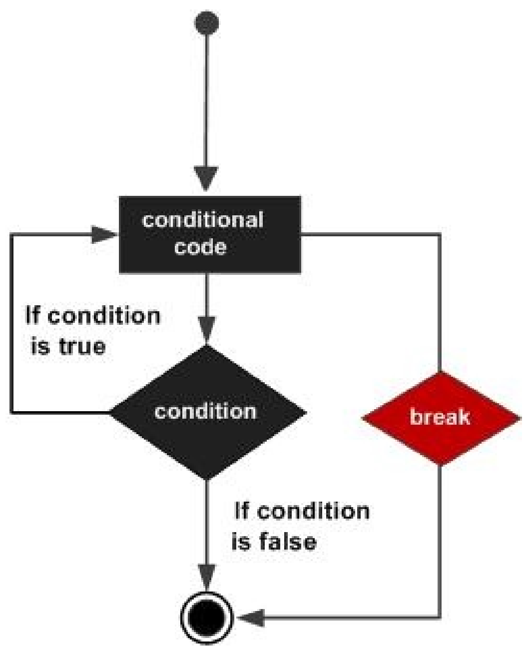
Scala 语言中默认是没有 break 语句，但是你在 Scala 2.8 版本后可以使用另外一种方式来实现 *break* 语句。当在循环中使用 **break** 语句，在执行到该语句时，就会中断循环并执行循环体之后的代码块。

语法

Scala 中 break 的语法有点不大一样，格式如下：

```
// 导入以下包
import scala.util.control._

// 创建 Breaks 对象
val loop = new Breaks;

// 在 breakable 中循环
loop.breakable{
    // 循环
    for(...){
       ....
       // 循环中断
       loop.break;
   }
}
```

流程图



实例

```
import scala.util.control._

object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      val loop = new Breaks;
      loop.breakable {
         for( a <- numList){
            println( "Value of a: " + a );
            if( a == 4 ){
               loop.break;
            }
         }
      }
      println( "After the loop" );
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a: 1
Value of a: 2
Value of a: 3
Value of a: 4
After the loop
```

中断嵌套循环

以下实例演示了如何中断嵌套循环：

```
import scala.util.control._

object Test {
   def main(args: Array[String]) {
      var a = 0;
      var b = 0;
      val numList1 = List(1,2,3,4,5);
      val numList2 = List(11,12,13);

      val outer = new Breaks;
      val inner = new Breaks;

      outer.breakable {
         for( a <- numList1){
            println( "Value of a: " + a );
            inner.breakable {
               for( b <- numList2){
                  println( "Value of b: " + b );
                  if( b == 12 ){
                     inner.break;
                  }
               }
            } // 内嵌循环中断
         }
      } // 外部循环中断
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a: 1
Value of b: 11
Value of b: 12
Value of a: 2
Value of b: 11
Value of b: 12
Value of a: 3
Value of b: 11
Value of b: 12
Value of a: 4
Value of b: 11
Value of b: 12
Value of a: 5
Value of b: 11
Value of b: 12
```

Scala 函数

函数是一组一起执行一个任务的语句。 您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

Scala 有函数和方法，二者在语义上的区别很小。Scala 方法是类的一部分，而函数是一个对象可以赋值给一个变量。换句话来说在类中定义的函数即是方法。

我们可以在任何地方定义函数，甚至可以在函数内定义函数（内嵌函数）。更重要的一点是 Scala 函数名可以由以下特殊字符：**+, ++, ~, &,-, --** , **\, /,** : 等。

函数声明

Scala 函数声明格式如下：

```
def functionName ([参数列表]) : [return type]
```

如果你不写等于号和方法主体，那么方法会被隐式声明为"抽象(abstract)"，包含它的类型于是也是一个抽象类型。

函数定义

方法定义由一个def 关键字开始， 紧接着是可选的参数列表，一个冒号"：" 和方法的返回类型，一个等于号"="，最后是方法的主体。

Scala 函数定义格式如下：

```
def functionName ([参数列表]) : [return type] = {
   function body
   return [expr]
}
```

以上代码中 **return type** 可以是任意合法的 Scala 数据类型。参数列表中的参数可以使用逗号分隔。

以下函数的功能是将两个传入的参数相加并求和：

```
object add{
   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

如果函数没有返回值，可以返回为 **Unit**，这个类似于 Java 的 **void**, 实例如下：

```
object Hello{
   def printMe( ) : Unit = {
      println("Hello, Scala!")
   }
}
```

函数调用

Scala 提供了多种不同的函数调用方式：

以下是调用方法的标准格式：

```
functionName( 参数列表 )
```

如果函数使用了实例的对象来调用，我们可以使用类似java的格式 (使用 **.** 号)：

```
[instance.]functionName( 参数列表 )
```

以上实例演示了定义与调用函数的实例:

```
object Test {
   def main(args: Array[String]) {
        println( "Returned Value : " + addInt(5,7) );
   }
   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Returned Value : 12
```

Scala也是一种函数式语言，所以函数是 Scala 语言的核心。以下一些函数概念有助于我们更好的理解 Scala 编程：

函数概念解析接案例	
函数传名调用(Call-by-Name)	指定函数参数名
函数 - 可变参数	递归函数
默认参数值	高阶函数
内嵌函数	匿名函数
偏应用函数	函数柯里化(Function Currying)

Scala 函数传名调用(call-by-name)

Scala的解释器在解析函数参数(function arguments)时有两种方式：

- 传值调用（call-by-value）：先计算参数表达式的值，再应用到函数内部；
- 传名调用（call-by-name）：将未计算的参数表达式直接应用到函数内部

在进入函数内部前，传值调用方式就已经将参数表达式的值计算完毕，而传名调用是在函数内部进行参数表达式的值计算的。

这就造成了一种现象，每次使用传名调用时，解释器都会计算一次表达式的值。

```
object Test {
   def main(args: Array[String]) {
        delayed(time());
   }

   def time() = {
      println("获取时间，单位为纳秒")
      System.nanoTime
   }
   def delayed( t: => Long ) = {
      println("在 delayed 方法内")
      println("参数： " + t)
      t
   }
}
```

以上实例中我们声明了 delayed 方法， 该方法在变量名和变量类型使用 => 符号来设置传名调用。执行以上代码，输出结果如下：

```
$ scalac Test.scala
$ scala Test
在 delayed 方法内
获取时间，单位为纳秒
参数： 241550840475831
获取时间，单位为纳秒
```

实例中 delay 方法打印了一条信息表示进入了该方法，接着 delay 方法打印接收到的值，最后再返回 t。

Scala 指定函数参数名

一般情况下函数调用参数，就按照函数定义时的参数顺序一个个传递。但是我们也可以通过指定函数参数名，并且不需要按照顺序向函数传递参数，实例如下：

```
object Test {
   def main(args: Array[String]) {
        printInt(b=5, a=7);
   }
   def printInt( a:Int, b:Int ) = {
      println("Value of a : " + a );
      println("Value of b : " + b );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a :  7
Value of b :  5
```

Scala 函数 - 可变参数

Scala 允许你指明函数的最后一个参数可以是重复的，即我们不需要指定函数参数的个数，可以向函数传入可变长度参数列表。

Scala 通过在参数的类型之后放一个星号来设置可变参数(可重复的参数)。例如：

```
object Test {
   def main(args: Array[String]) {
        printStrings("Runoob", "Scala", "Python");
   }
   def printStrings( args:String* ) = {
      var i : Int = 0;
      for( arg <- args ){
         println("Arg value[" + i + "] = " + arg );
         i = i + 1;
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Arg value[0] = Runoob
Arg value[1] = Scala
Arg value[2] = Python
```

Scala 递归函数

递归函数在函数式编程的语言中起着重要的作用。

Scala 同样支持递归函数。

递归函数意味着函数可以调用它本身。

以上实例使用递归函数来计算阶乘：

```
object Test {
   def main(args: Array[String]) {
      for (i <- 1 to 10)
         println(i + " 的阶乘为: = " + factorial(i) )
   }

   def factorial(n: BigInt): BigInt = {
      if (n <= 1)
         1
      else
      n * factorial(n - 1)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1 的阶乘为: = 1
2 的阶乘为: = 2
3 的阶乘为: = 6
4 的阶乘为: = 24
5 的阶乘为: = 120
6 的阶乘为: = 720
7 的阶乘为: = 5040
8 的阶乘为: = 40320
9 的阶乘为: = 362880
10 的阶乘为: = 3628800
```

Scala 高阶函数

高阶函数（Higher-Order Function）就是操作其他函数的函数。

Scala 中允许使用高阶函数, 高阶函数可以使用其他函数作为参数，或者使用函数作为输出结果。

以下实例中，apply() 函数使用了另外一个函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v：

```
object Test {
   def main(args: Array[String]) {

      println( apply( layout, 10) )

   }
   // 函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v
   def apply(f: Int => String, v: Int) = f(v)

   def layout[A](x: A) = "[" + x.toString() + "]"

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
[10]
```

Scala 函数嵌套

我么可以在 Scala 函数内定义函数，定义在函数内的函数称之为局部函数。

以下实例我们实现阶乘运算，并使用内嵌函数：

```
object Test {
   def main(args: Array[String]) {
      println( factorial(0) )
      println( factorial(1) )
      println( factorial(2) )
      println( factorial(3) )
   }

   def factorial(i: Int): Int = {
      def fact(i: Int, accumulator: Int): Int = {
         if (i <= 1)
            accumulator
         else
            fact(i - 1, i * accumulator)
      }
      fact(i, 1)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1
1
2
6
```

Scala 匿名函数

Scala 中定义匿名函数的语法很简单，箭头左边是参数列表，右边是函数体，参数的类型是可省略的，Scala 的类型推测系统会推测出参数的类型。使用匿名函数后，我们的代码变得更简洁了。

下面的表达式就定义了一个接受一个Int类型输入参数的匿名函数:

```
var inc = (x:Int) => x+1
```

上述定义的匿名函数，其实是下面这种写法的简写：

```
def add2 = new Function1[Int,Int]{
    def apply(x:Int):Int = x+1;
}
```

以上实例的 inc 现在可作为一个函数，使用方式如下：

```
var x = inc(7)-1
```

同样我们可以在匿名函数中定义多个参数：

```
var mul = (x: Int, y: Int) => x*y
```

mul 现在可作为一个函数，使用方式如下：

```
println(mul(3, 4))
```

我们也可以不给匿名函数设置参数，如下所示：

```
var userDir = () => { System.getProperty("user.dir") }
```

userDir 现在可作为一个函数，使用方式如下：

```
println( userDir )
```

Scala 偏应用函数

Scala 偏应用函数是一种表达式，你不需要提供函数需要的所有参数，只需要提供部分，或不提供所需参数。

如下实例，我们打印日志信息：

```
import java.util.Date

object Test {
   def main(args: Array[String]) {
      val date = new Date
      log(date, "message1" )
      Thread.sleep(1000)
      log(date, "message2" )
      Thread.sleep(1000)
      log(date, "message3" )
   }

   def log(date: Date, message: String)  = {
     println(date + "----" + message)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Mon Dec 02 12:52:41 CST 2013----message1
Mon Dec 02 12:52:41 CST 2013----message2
Mon Dec 02 12:52:41 CST 2013----message3
```

实例中，log() 方法接收两个参数：date 和 message。我们在程序执行时调用了三次，参数 date 值都相同，message 不同。

我们可以使用偏应用函数优化以上方法，绑定第一个 date 参数，第二个参数使用下划线(_)替换缺失的参数列表，并把这个新的函数值的索引的赋给变量。以上实例修改如下：

```
import java.util.Date

object Test {
   def main(args: Array[String]) {
      val date = new Date
      val logWithDateBound = log(date, _ : String)

      logWithDateBound("message1" )
      Thread.sleep(1000)
      logWithDateBound("message2" )
      Thread.sleep(1000)
      logWithDateBound("message3" )
   }

   def log(date: Date, message: String)  = {
     println(date + "----" + message)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Mon Dec 02 12:53:56 CST 2013----message1
Mon Dec 02 12:53:56 CST 2013----message2
Mon Dec 02 12:53:56 CST 2013----message3
```

Scala 函数柯里化(Currying)

柯里化(Currying)指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数。

实例

首先我们定义一个函数:

```
def add(x:Int,y:Int)=x+y
```

那么我们应用的时候，应该是这样用：add(1,2)

现在我们把这个函数变一下形：

```
def add(x:Int)(y:Int)  = x + y
```

那么我们应用的时候，应该是这样用：add(1)(2),最后结果都一样是3，这种方式（过程）就叫柯里化。

实现过程

add(1)(2) 实际上是依次调用两个普通函数（非柯里化函数），第一次调用使用一个参数 x，返回一个函数类型的值，第二次使用参数y调用这个函数类型的值。

实质上最先演变成这样一个方法：

```
def add(x:Int)=(y:Int)=>x+y
```

那么这个函数是什么意思呢？ 接收一个x为参数，返回一个匿名函数，该匿名函数的定义是：接收一个Int型参数y，函数体为x+y。现在我们来对这个方法进行调用。

```
val result = add(1)
```

返回一个result，那result的值应该是一个匿名函数：(y:Int)=>1+y

所以为了得到结果，我们继续调用result。

```
val sum = result(2)
```

最后打印出来的结果就是3。

完整实例

下面是一个完整实例：

```
object Test {
   def main(args: Array[String]) {
      val str1:String = "Hello, "
      val str2:String = "Scala!"
      println( "str1 + str2 = " +  strcat(str1)(str2) )
   }

   def strcat(s1: String)(s2: String) = {
      s1 + s2
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
str1 + str2 = Hello, Scala!
```

Scala 闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

闭包通常来讲可以简单的认为是可以访问一个函数里面局部变量的另外一个函数。

如下面这段匿名的函数：

```
val multiplier = (i:Int) => i * 10
```

函数体内有一个变量 i，它作为函数的一个参数。如下面的另一段代码：

```
val multiplier = (i:Int) => i * factor
```

在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形式参数，在 multiplier 函数被调用时，i 被赋予一个新的值。然而，factor不是形式参数，而是自由变量，考虑下面代码：

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

这里我们引入一个自由变量 factor，这个变量定义在函数外面。

这样定义的函数变量 multiplier 成为一个"闭包"，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数。

完整实例

```
object Test {
   def main(args: Array[String]) {
      println( "muliplier(1) value = " +  multiplier(1) )
      println( "muliplier(2) value = " +  multiplier(2) )
   }
   var factor = 3
   val multiplier = (i:Int) => i * factor
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$  scala Test
muliplier(1) value = 3
muliplier(2) value = 6
```

Scala 字符串

以下实例将字符串赋值给一个常量：

```
object Test {
   val greeting: String = "Hello,World!"

   def main(args: Array[String]) {
      println( greeting )
   }
}
```

以上实例定义了变量 greeting，为字符串常量，它的类型为 String (java.lang.String)。

在 Scala 中，字符串的类型实际上是 Java String，它本身没有 String 类。

在 Scala 中，String 是一个不可变的对象，所以该对象不可被修改。这就意味着你如果修改字符串就会产生一个新的字符串对象。

但其他对象，如数组就是可变的对象。接下来我们会为大家介绍常用的 java.lang.String 方法。

创建字符串

创建字符串实例如下：

```
var greeting = "Hello World!";

或

var greeting:String = "Hello World!";
```

你不一定为字符串指定 String 类型，因为 Scala 编译器会自动推断出字符串的类型为 String。

当然我们也可以直接显示的声明字符串为 String 类型，如下实例：

```
object Test {
   val greeting: String = "Hello, World!"

   def main(args: Array[String]) {
      println( greeting )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Hello, world!
```

我们前面提到过 String 对象是不可变的，如果你需要创建一个可以修改的字符串，可以使用 String Builder 类，如下实例:

```
object Test {
   def main(args: Array[String]) {
      val buf = new StringBuilder;
      buf += 'a'
      buf ++= "bcdef"
      println( "buf is : " + buf.toString );
   }
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
buf is : abcdef
```

字符串长度

我们可以使用 length() 方法来获取字符串长度：

```
object Test {
   def main(args: Array[String]) {
      var palindrome = "www.runoob.com";
      var len = palindrome.length();
      println( "String Length is : " + len );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
String Length is : 14
```

字符串连接

String 类中使用 concat() 方法来连接两个字符串：

```
string1.concat(string2);
```

实例演示：

```
scala> "菜鸟教程官网： ".concat("www.runoob.com");
res0: String = 菜鸟教程官网： www.runoob.com
```

同样你也可以使用加号(+)来连接：

```
scala> "菜鸟教程官网： " + " www.runoob.com"
res1: String = 菜鸟教程官网：  www.runoob.com
```

让我们看个完整实例:

```
object Test {
   def main(args: Array[String]) {
      var str1 = "菜鸟教程官网：";
      var str2 =  "www.runoob.com";
      var str3 =  "菜鸟教程的 Slogan 为：";
      var str4 =  "学的不仅是技术，更是梦想！";
      println( str1 + str2 );
      println( str3.concat(str4) );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
菜鸟教程官网：www.runoob.com
菜鸟教程的 Slogan 为：学的不仅是技术，更是梦想！
```

创建格式化字符串

String 类中你可以使用 printf() 方法来格式化字符串并输出，String format() 方法可以返回 String 对象而不是 PrintStream 对象。以下实例演示了 printf() 方法的使用：

```
object Test {
   def main(args: Array[String]) {
      var floatVar = 12.456
      var intVar = 2000
      var stringVar = "菜鸟教程!"
      var fs = printf("浮点型变量为 " +
                   "%f, 整型变量为 %d, 字符串为 " +
                   " %s", floatVar, intVar, stringVar)
      println(fs)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
浮点型变量为 12.456000, 整型变量为 2000, 字符串为  菜鸟教程!()
```

String 方法

下表列出了 java.lang.String 中常用的方法，你可以在 Scala 中使用：

方法	描述
char charAt(int index)	返回指定位置的字符
int compareTo(Object o)	比较字符串与对象
int compareTo(String anotherString)	按字典顺序比较两个字符串
int compareToIgnoreCase(String str)	按字典顺序比较两个字符串，不考虑大小写
String concat(String str)	将指定字符串连接到此字符串的结尾
boolean contentEquals(StringBuffer sb)	将此字符串与指定的 StringBuffer 比较。
static String copyValueOf(char[] data)	返回指定数组中表示该字符序列的 String
static String copyValueOf(char[] data, int offset, int count)	返回指定数组中表示该字符序列的 String
boolean endsWith(String suffix)	测试此字符串是否以指定的后缀结束
boolean equals(Object anObject)	将此字符串与指定的对象比较
boolean equalsIgnoreCase(String anotherString)	将此 String 与另一个 String 比较，不考虑大小写
byte getBytes()	使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中
byte[] getBytes(String charsetName	使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	将字符从此字符串复制到目标字符数组
int hashCode()	返回此字符串的哈希码
int indexOf(int ch)	返回指定字符在此字符串中第一次出现处的索引
int indexOf(int ch, int fromIndex)	返返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索
int indexOf(String str)	返回指定子字符串在此字符串中第一次出现处的索引

int indexOf(String str, int fromIndex)	返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
String intern()	返回字符串对象的规范化表示形式
int lastIndexOf(int ch)	返回指定字符在此字符串中最后一次出现处的索引
int lastIndexOf(int ch, int fromIndex)	返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索
int lastIndexOf(String str)	返回指定子字符串在此字符串中最右边出现处的索引
int lastIndexOf(String str, int fromIndex)	返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索
int length()	返回此字符串的长度
boolean matches(String regex)	告知此字符串是否匹配给定的正则表达式
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等
boolean regionMatches(int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等
String replace(char oldChar, char newChar)	返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的
String replaceAll(String regex, String replacement	使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串
String replaceFirst(String regex, String replacement)	使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串
String[] split(String regex)	根据给定正则表达式的匹配拆分此字符串
String[] split(String regex, int limit)	根据匹配给定的正则表达式来拆分此字符串
boolean startsWith(String prefix)	测试此字符串是否以指定的前缀开始
boolean startsWith(String prefix, int toffset)	测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
CharSequence subSequence(int beginIndex, int endIndex)	返回一个新的字符序列，它是此序列的一个子序列
String substring(int beginIndex)	返回一个新的字符串，它是此字符串的一个子字符串
String substring(int beginIndex, int endIndex)	返回一个新字符串，它是此字符串的一个子字符串

char[] toCharArray()	将此字符串转换为一个新的字符数组
String toLowerCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为小写
String toLowerCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为小写
String toString()	返回此对象本身（它已经是一个字符串！）
String toUpperCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为大写
String toUpperCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为大写
String trim()	使用给定 Locale 的规则将此 String 中的所有字符都转换为大写
static String valueOf(primitive data type x)	返回指定类型参数的字符串表示形式

Scala 数组

Scala 语言中提供的数组是用来存储固定大小的同类型元素，数组对于每一门编辑应语言来说都是重要的数据结构之一。

声明数组变量并不是声明 number0、number1、...、number99 一个个单独的变量，而是声明一个就像 numbers 这样的变量，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

数组的第一个元素索引为0，最后一个元素的索引为元素总数减1。

声明数组

以下是 Scala 数组声明的语法格式：

```
var z:Array[String] = new Array[String](3)

或

var z = new Array[String](3)
```

以上语法中，z 声明一个字符串类型的数组，数组长度为 3 ，可存储 3 个元素。我们可以为每个元素设置值，并通过索引来访问每个元素，如下所示：

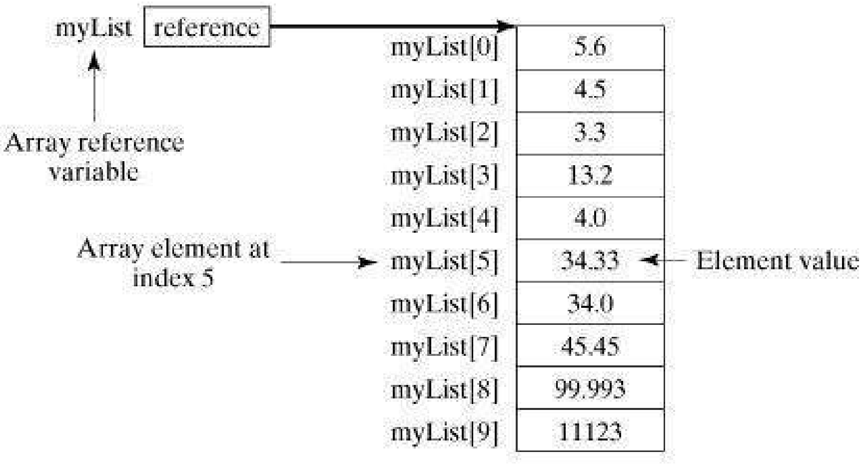
```
z(0) = "Runoob"; z(1) = "Baidu"; z(4/2) = "Google"
```

最后一个元素的索引使用了表达式 **4/2** 作为索引，类似于 **z(2) = "Google"**。

我们也可以使用以下方式来定义一个数组：

```
var z = Array("Runoob", "Baidu", "Google")
```

下图展示了一个长度为 10 的数组 myList，索引值为 0 到 9：



处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本的 for 循环。

以下实例演示了数组的创建，初始化等处理过程：

```
object Test {
   def main(args: Array[String]) {
      var myList = Array(1.9, 2.9, 3.4, 3.5)

      // 输出所有数组元素
      for ( x <- myList ) {
         println( x )
      }

      // 计算数组所有元素的总会
      var total = 0.0;
      for ( i <- 0 to (myList.length - 1)) {
         total += myList(i);
      }
      println("总和为 " + total);

      // 查找数组中的最大元素
      var max = myList(0);
      for ( i <- 1 to (myList.length - 1) ) {
         if (myList(i) > max) max = myList(i);
      }
      println("最大值为 " + max);

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1.9
2.9
3.4
3.5
总和为 11.7
最大值为 3.5
```

多维数组

多维数组一个数组中的值可以是另一个数组，另一个数组的值也可以是一个数组。矩阵与表格是我们常见的二维数组。

以上是一个定义了二维数组的实例：

```
var myMatrix = ofDim[Int](3,3)
```

实例中数组中包含三个数组元素，每个数组元素又含有三个值。

接下来我们来看一个二维数组处理的完整实例：

```
import Array._

object Test {
   def main(args: Array[String]) {
      var myMatrix = ofDim[Int](3,3)
      
      // 创建矩阵
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            myMatrix(i)(j) = j;
         }
      }
      
      // 打印二维阵列
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            print(" " + myMatrix(i)(j));
         }
         println();
      }
    
   }
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
0 1 2
0 1 2
0 1 2
```

合并数组

以下实例中，我们使用 concat() 方法来合并两个数组，concat() 方法中接受多个数组参数：

```
import Array._

object Test {
   def main(args: Array[String]) {
      var myList1 = Array(1.9, 2.9, 3.4, 3.5)
      var myList2 = Array(8.9, 7.9, 0.4, 1.5)

      var myList3 =  concat( myList1, myList2)

      // 输出所有数组元素
      for ( x <- myList3 ) {
         println( x )
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1.9
2.9
3.4
3.5
8.9
7.9
0.4
1.5
```

创建区间数组

以下实例中，我们使用了 range() 方法来生成一个区间范围内的数组。range() 方法最后一个参数为步长，默认为 1：

```
import Array._

object Test {
   def main(args: Array[String]) {
      var myList1 = range(10, 20, 2)
      var myList2 = range(10,20)

      // 输出所有数组元素
      for ( x <- myList1 ) {
         print( " " + x )
      }
      println()
      for ( x <- myList2 ) {
         print( " " + x )
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
10 12 14 16 18
10 11 12 13 14 15 16 17 18 19
```

Scala 数组方法

下表中为 Scala 语言中处理数组的重要方法，使用它前我们需要使用 **import Array._** 引入包。

方法	描述
def apply(x: T, xs: T*): Array[T]	创建指定对象 T 的数组, T 的值可以是 Unit, Double, Float, Long Boolean。
def concat[T](xss: Array[T]*): Array[T]	合并数组
def copy(src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int): Unit	复制一个数组到另一个数组上。相等于 Java's System.arraycop destPos, length)。
def empty[T]: Array[T]	返回长度为 0 的数组
def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]	返回指定长度数组，每个数组元素为指定函数的返回值。以上实为 3，计算函数为**a=>a+1**： `scala> Array.iterate(0,3)(a=>a+1) res1: Arr`
def fill[T](n: Int)(elem: => T): Array[T]	返回数组，长度为第一个参数指定，同时每个元素使用第二个参
def fill[T](n1: Int, n2: Int)(elem: => T): Array[Array[T]]	返回二数组，长度为第一个参数指定，同时每个元素使用第二个
def ofDim[T](n1: Int): Array[T]	创建指定长度的数组
def ofDim[T](n1: Int, n2: Int): Array[Array[T]]	创建二维数组
def ofDim[T](n1: Int, n2: Int, n3: Int): Array[Array[Array[T]]]	创建三维数组
def range(start: Int, end: Int, step: Int): Array[Int]	创建指定区间内的数组，step 为每个元素间的步长
def range(start: Int, end: Int): Array[Int]	创建指定区间内的数组
def tabulate[T](n: Int)(f: (Int)=> T): Array[T]	返回指定长度数组，每个数组元素为指定函数的返回值，默认从个元素： `scala> Array.tabulate(3)(a => a + 5) res0: Arr`
def tabulate[T](n1: Int, n2: Int)(f: (Int, Int) => T): Array[Array[T]]	返回指定长度的二维数组，每个数组元素为指定函数的返回值，

Scala Collection

Scala提供了一套很好的集合实现，提供了一些集合类型的抽象。

Scala 集合分为可变的和不可变的集合。

可变集合可以在适当的地方被更新或扩展。这意味着你可以修改，添加，移除一个集合的元素。

而不可变集合类，相比之下，永远不会改变。不过，你仍然可以模拟添加，移除或更新操作。但是这些操作将在每一种情况下都返回一个新的集合，同时使原来的集合不发生改变。

接下来我们将为大家介绍几种常用集合类型的应用：

序号	集合	描述
1	Scala List(列表)	List的特征是其元素以线性方式存储，集合中可以存放重复对象。 参考 API文档
2	Scala Set(集合)	Set是最简单的一种集合。集合中的对象不按特定的方式排序，并且没有重复对象。 参考 API文档
3	Scala Map(映射)	Map 是一种把键对象和值对象映射的集合，它的每一个元素都包含一对键对象和值对象。 参考 API文档
4	Scala 元组	元组是不同类型的值的集合
5	Scala Option	Option[T] 表示有可能包含值的容器，也可能不包含值。
6	Scala Iterator（迭代器）	迭代器不是一个容器，更确切的说是逐一访问容器内元素的方法。

实例

以下代码判断，演示了所有以上集合类型的定义实例：

```
// 定义整型 List
val x = List(1,2,3,4)

// 定义 Set
var x = Set(1,3,5,7)

// 定义 Map
val x = Map("one" -> 1, "two" -> 2, "three" -> 3)

// 创建两个不同类型元素的元组
val x = (10, "Runoob")

// 定义 Option
val x:Option[Int] = Some(5)
```

Scala List(列表)

Scala 列表类似于数组，它们所有元素的类型都相同，但是它们也有所不同：列表是不可变的，值一旦被定义了就不能改变，其次列表 具有递归的结构（也就是链接表结构）而数组不是。。

列表的元素类型 T 可以写成 List[T]。例如，以下列出了多种类型的列表：

```
// 字符串列表
val site: List[String] = List("Runoob", "Google", "Baidu")

// 整型列表
val nums: List[Int] = List(1, 2, 3, 4)

// 空列表
val empty: List[Nothing] = List()

// 二维列表
val dim: List[List[Int]] =
   List(
      List(1, 0, 0),
      List(0, 1, 0),
      List(0, 0, 1)
   )
```

构造列表的两个基本单位是 **Nil** 和 **::**

Nil 也可以表示为一个空列表。

以上实例我们可以写成如下所示：

```
// 字符串列表
val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

// 整型列表
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// 空列表
val empty = Nil

// 二维列表
val dim = (1 :: (0 :: (0 :: Nil))) ::
          (0 :: (1 :: (0 :: Nil))) ::
          (0 :: (0 :: (1 :: Nil))) :: Nil
```

列表基本操作

Scala列表有三个基本操作：

- `head` 返回列表第一个元素
- `tail` 返回一个列表，包含除了第一元素之外的其他元素
- `isEmpty` 在列表为空时返回true

对于Scala列表的任何操作都可以使用这三个基本操作来表达。实例如下:

```
object Test {
   def main(args: Array[String]) {
      val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      val nums = Nil

      println( "第一网站是 : " + site.head )
      println( "最后一个网站是 : " + site.tail )
      println( "查看列表 site 是否为空 : " + site.isEmpty )
      println( "查看 nums 是否为空 : " + nums.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
第一网站是 : Runoob
最后一个网站是 : List(Google, Baidu)
查看列表 site 是否为空 : false
查看 nums 是否为空 : true
```

连接列表

你可以使用 **:::** 运算符或 **List.:::()** 方法或 **List.concat()** 方法来连接两个或多个列表。实例如下:

```
object Test {
   def main(args: Array[String]) {
      val site1 = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      val site2 = "Facebook" :: ("Taobao" :: Nil)

      // 使用 ::: 运算符
      var fruit = site1 ::: site2
      println( "site1 ::: site2 : " + fruit )

      // 使用 Set.:::() 方法
      fruit = site1.:::(site2)
      println( "site1.:::(site2) : " + fruit )

      // 使用 concat 方法
      fruit = List.concat(site1, site2)
      println( "List.concat(site1, site2) : " + fruit  )


   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site1 ::: site2 : List(Runoob, Google, Baidu, Facebook, Taobao)
site1.:::(site2) : List(Facebook, Taobao, Runoob, Google, Baidu)
List.concat(site1, site2) : List(Runoob, Google, Baidu, Facebook, Taobao)
```

List.fill()

我们可以使用 List.fill() 方法来创建一个指定重复数量的元素列表：

```
object Test {
   def main(args: Array[String]) {
      val site = List.fill(3)("Runoob") // 重复 Runoob 3次
      println( "site : " + site  )

      val num = List.fill(10)(2)         // 重复元素 2, 10 次
      println( "num : " + num  )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site : List(Runoob, Runoob, Runoob)
num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
```

List.tabulate()

List.tabulate() 方法是通过给定的函数来创建列表。

方法的第一个参数为元素的数量，可以是二维的，第二个参数为指定的函数，我们通过指定的函数计算结果并返回值插入到列表中，起始值为 0，实例如下：

```
object Test {
   def main(args: Array[String]) {
      // 通过给定的函数创建 5 个元素
      val squares = List.tabulate(6)(n => n * n)
      println( "一维 : " + squares  )

      // 创建二维列表
      val mul = List.tabulate( 4,5 )( _ * _ )
      println( "多维 : " + mul  )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
一维 : List(0, 1, 4, 9, 16, 25)
多维 : List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8), List(0, 3, 6, 9
```

List.reverse

List.reverse 用于将列表的顺序反转，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      println( "site 反转前 : " + site )

      println( "site 反转前 : " + site.reverse )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site 反转前 : List(Runoob, Google, Baidu)
site 反转前 : List(Baidu, Google, Runoob)
```

Scala List 常用方法

下表列出了 Scala List 常用的方法：

方法	描述
def +(elem: A): List[A]	为列表预添加元素
def ::(x: A): List[A]	在列表开头添加元素
def :::(prefix: List[A]): List[A]	在列表开头添加指定列表的元素
def ::(x: A): List[A]	在列表开头添加元素 x
def addString(b: StringBuilder): StringBuilder	将列表的所有元素添加到 StringBuilder
def addString(b: StringBuilder, sep: String): StringBuilder	将列表的所有元素添加到 StringBuilder，并指定分隔符
def apply(n: Int): A	通过列表索引获取元素
def contains(elem: Any): Boolean	检测列表中是否包含指定的元素
def copyToArray(xs:	

Array[A], start: Int, len: Int): Unit	
def distinct: List[A]	去除列表的重复元素，并返回新列表
def drop(n: Int): List[A]	丢弃前n个元素，并返回新列表
def dropRight(n: Int): List[A]	丢弃最后n个元素，并返回新列表
def dropWhile(p: (A) => Boolean): List[A]	从左向右丢弃元素，直到条件p不成立
def endsWith[B](that: Seq[B]): Boolean	检测列表是否以指定序列结尾
def equals(that: Any): Boolean	判断是否相等
def exists(p: (A) => Boolean): Boolean	判断列表中指定条件的元素是否存在。判断l是否存在某个元素: `scala> l.exists(s => s == "Hah") res7: Boolean = true`
def filter(p: (A) => Boolean): List[A]	输出符号指定条件的所有元素。过滤出长度为3的元素: `scala> l.filter(s => s.length == 3) res8: List[String] =`
def forall(p: (A) => Boolean): Boolean	检测所有元素。例如：判断所有元素是否以"H"开头： `scala> l.forall(s => s.startsWith("H")) res10: Boolean`
def foreach(f: (A) => Unit): Unit	将函数应用到列表的所有元素
def head: A	获取列表的第一个元素
def indexOf(elem: A, from: Int): Int	从指定位置 from 开始查找元素第一次出现的位置
def init: List[A]	返回所有元素，除了最后一个
def intersect(that: Seq[A]): List[A]	计算多个集合的交集
def isEmpty: Boolean	检测列表是否为空
def iterator: Iterator[A]	创建一个新的迭代器来迭代元素
def last: A	返回最后一个元素
def lastIndexOf(elem:	在指定的位置 end 开始查找元素最后出现的位置

lastIndexOf(elem: A, end: Int): Int	在指定的位置 end 开始查找元素最后出现的位置
def length: Int	返回列表长度
def map[B](f: (A) => B): List[B]	通过给定的方法将所有元素重新计算
def max: A	查找最大元素
def min: A	查找最小元素
def mkString: String	列表所有元素作为字符串显示
def mkString(sep: String): String	使用分隔符将列表所有元素作为字符串显示
def reverse: List[A]	列表反转
def sorted[B >: A]: List[A]	列表排序
def startsWith[B](that: Seq[B], offset: Int): Boolean	检测列表在指定位置是否包含指定序列
def sum: A	计算集合元素之和
def tail: List[A]	返回所有元素，除了第一个
def take(n: Int): List[A]	提取列表的前n个元素
def takeRight(n: Int): List[A]	提取列表的后n个元素
def toArray: Array[A]	列表转换为数组
def toBuffer[B >: A]: Buffer[B]	返回缓冲区，包含了列表的所有元素
def toMap[T, U]: Map[T, U]	List 转换为 Map
def toSeq: Seq[A]	List 转换为 Seq
def toSet[B >: A]: Set[B]	List 转换为 Set
def toString(): String	列表转换为字符串

更多方法可以参考 API文档

Scala Set(集合)

Scala Set(集合)是没有重复的对象集合，所有的元素都是唯一的。

Scala 集合分为可变的和不可变的集合。

默认情况下，Scala 使用的是不可变集合，如果你想使用可变集合，需要引用 **scala.collection.mutable.Set** 包。

默认引用 scala.collection.immutable.Set，不可变集合实例如下：

```
val set = Set(1,2,3)
println(set.getClass.getName) //

println(set.exists(_ % 2 == 0)) //true
println(set.drop(1)) //Set(2,3)
```

如果需要使用可变集合需要引入 scala.collection.mutable.Set：

```
import scala.collection.mutable.Set // 可以在任何地方引入 可变集合

val mutableSet = Set(1,2,3)
println(mutableSet.getClass.getName) // scala.collection.mutable.HashSet

mutableSet.add(4)
mutableSet.remove(1)
mutableSet += 5
mutableSet -= 2

println(mutableSet) // Set(5, 3, 4)

val another = mutableSet.toSet
println(another.getClass.getName) // scala.collection.immutable.Set
```

> 注意： 虽然可变Set和不可变Set都有添加或删除元素的操作，但是有一个非常大的差别。对不可变Set进行操作，会产生一个新的set，原来的set并没有改变，这与List一样。而对可变Set进行操作，改变的是该Set本身，与ListBuffer类似。

集合基本操作

Scala集合有三个基本操作：

- `head` 返回集合第一个元素
- `tail` 返回一个集合，包含除了第一元素之外的其他元素
- `isEmpty` 在集合为空时返回true

对于Scala集合的任何操作都可以使用这三个基本操作来表达。实例如下:

```
object Test {
   def main(args: Array[String]) {
      val site = Set("Runoob", "Google", "Baidu")
      val nums: Set[Int] = Set()

      println( "第一网站是 : " + site.head )
      println( "最后一个网站是 : " + site.tail )
      println( "查看列表 site 是否为空 : " + site.isEmpty )
      println( "查看 nums 是否为空 : " + nums.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
第一网站是 : Runoob
最后一个网站是 : Set(Google, Baidu)
查看列表 site 是否为空 : false
查看 nums 是否为空 : true
```

连接集合

你可以使用 **++** 运算符或 **Set.++()** 方法来连接两个集合。如果元素有重复的就会移除重复的元素。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val site1 = Set("Runoob", "Google", "Baidu")
      val site2 = Set("Faceboook", "Taobao")

      // ++ 作为运算符使用
      var site = site1 ++ site2
      println( "site1 ++ site2 : " + site )

      //  ++ 作为方法使用
      site = site1.++(site2)
      println( "site1.++(site2) : " + site )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site1 ++ site2 : Set(Faceboook, Taobao, Google, Baidu, Runoob)
site1.++(site2) : Set(Faceboook, Taobao, Google, Baidu, Runoob)
```

查找集合中最大与最小元素

你可以使用 **Set.min** 方法来查找集合中的最小元素，使用 **Set.max** 方法查找集合中的最大元素。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val num = Set(5,6,9,20,30,45)

      // 查找集合中最大与最小元素
      println( "Set(5,6,9,20,30,45) 集合中的最小元素是 : " + num.min )
      println( "Set(5,6,9,20,30,45) 集合中的最大元素是 : " + num.max )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
Set(5,6,9,20,30,45) 集合中的最小元素是 : 5
Set(5,6,9,20,30,45) 集合中的最大元素是 : 45
```

交集

你可以使用 **Set.&** 方法或 **Set.intersect** 方法来查看两个集合的交集元素。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val num1 = Set(5,6,9,20,30,45)
      val num2 = Set(50,60,9,20,35,55)

      // 交集
      println( "num1.&(num2) : " + num1.&(num2) )
      println( "num1.intersect(num2) : " + num1.intersect(num2) )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
num1.&(num2) : Set(20, 9)
num1.intersect(num2) : Set(20, 9)
```

Scala Set 常用方法

下表列出了 Scala Set 常用的方法：

方法	描述
def +(elem: A): Set[A]	为集合添加新元素，x并创建一个新的集合，除非元素已存在
def -(elem: A): Set[A]	移除集合中的元素，并创建一个新的集合
def contains(elem: A): Boolean	如果元素在集合中存在，返回 true，否则返回 false。

def &(that: Set[A]): Set[A]	返回两个集合的交集
def &~(that: Set[A]): Set[A]	返回两个集合的差集
def +(elem1: A, elem2: A, elems: A*): Set[A]	通过添加传入指定集合的元素创建一个新的不可变集合
def ++(elems: A): Set[A]	合并两个集合
def -(elem1: A, elem2: A, elems: A*): Set[A]	通过移除传入指定集合的元素创建一个新的不可变集合
def addString(b: StringBuilder): StringBuilder	将不可变集合的所有元素添加到字符串缓冲区
def addString(b: StringBuilder, sep: String): StringBuilder	将不可变集合的所有元素添加到字符串缓冲区，并使用指定的分隔符
def apply(elem: A)	检测集合中是否包含指定元素
def count(p: (A) => Boolean): Int	计算满足指定条件的集合元素个数
def copyToArray(xs: Array[A], start: Int, len: Int): Unit	复制不可变集合元素到数组
def diff(that: Set[A]): Set[A]	比较两个集合的差集
def drop(n: Int): Set[A]]	返回丢弃前n个元素新集合
def dropRight(n: Int): Set[A]	返回丢弃最后n个元素新集合
def dropWhile(p: (A) => Boolean): Set[A]	从左向右丢弃元素，直到条件p不成立
def equals(that: Any): Boolean	equals 方法可用于任意序列。用于比较系列是否相等。
def exists(p: (A) => Boolean): Boolean	判断不可变集合中指定条件的元素是否存在。
def filter(p: (A) => Boolean): Set[A]	输出符合指定条件的所有不可变集合元素。
def find(p: (A) => Boolean): Option[A]	查找不可变集合中满足指定条件的第一个元素
def forall(p: (A) => Boolean): Boolean	查找不可变集合中满足指定条件的所有元素
def foreach(f: (A) => Unit): Unit	将函数应用到不可变集合的所有元素
def head: A	获取不可变集合的第一个元素
def init: Set[A]	返回所有元素，除了最后一个
def intersect(that: Set[A]): Set[A]	计算两个集合的交集
def isEmpty: Boolean	判断集合是否为空
def iterator: Iterator[A]	创建一个新的迭代器来迭代元素

def last: A	返回最后一个元素
def map[B](f: (A) => B): immutable.Set[B]	通过给定的方法将所有元素重新计算
def max: A	查找最大元素
def min: A	查找最小元素
def mkString: String	集合所有元素作为字符串显示
def mkString(sep: String): String	使用分隔符将集合所有元素作为字符串显示
def product: A	返回不可变集合中数字元素的积。
def size: Int	返回不可变集合元素的数量
def splitAt(n: Int): (Set[A], Set[A])	把不可变集合拆分为两个容器，第一个由前 n 个元素组成，第二个由剩下的元素组成
def subsetOf(that: Set[A]): Boolean	如果集合中含有子集返回 true，否则返回false
def sum: A	返回不可变集合中所有数字元素之和
def tail: Set[A]	返回一个不可变集合中除了第一元素之外的其他元素
def take(n: Int): Set[A]	返回前 n 个元素
def takeRight(n: Int):Set[A]	返回后 n 个元素
def toArray: Array[A]	将集合转换为数字
def toBuffer[B >: A]: Buffer[B]	返回缓冲区，包含了不可变集合的所有元素
def toList: List[A]	返回 List，包含了不可变集合的所有元素
def toMap[T, U]: Map[T, U]	返回 Map，包含了不可变集合的所有元素
def toSeq: Seq[A]	返回 Seq，包含了不可变集合的所有元素
def toString(): String	返回一个字符串，以对象来表示

更多方法可以参考 API文档

Scala Map(映射)

Map(映射)是一种可迭代的键值对（key/value）结构。

所有的值都可以通过键来获取。

Map 中的键都是唯一的。

Map 也叫哈希表（Hash tables）。

Map 有两种类型，可变与不可变，区别在于可变对象可以修改它，而不可变对象不可以。

默认情况下 Scala 使用不可变 Map。如果你需要使用可变集合，你需要显式的引入 **import scala.collection.mutable.Map** 类

在 Scala 中 你可以同时使用可变与不可变 Map，不可变的直接使用 Map，可变的使用 mutable.Map。以下实例演示了不可变 Map 的应用：

```
// 空哈希表，键为字符串，值为整型
var A:Map[Char,Int] = Map()

// Map 键值对演示
val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")
```

定义 Map 时，需要为键值对定义类型。如果需要添加 key-value 对，可以使用 + 号，如下所示：

```
A += ('I' -> 1)
A += ('J' -> 5)
A += ('K' -> 10)
A += ('L' -> 100)
```

Map 基本操作

Scala Map 有三个基本操作：

方法	描述
keys	返回 Map 所有的键(key)
values	返回 Map 所有的值(value)
isEmpty	在 Map 为空时返回true

实例

以下实例演示了以上三个方法的基本应用：

```
object Test {
   def main(args: Array[String]) {
      val colors = Map("red" -> "#FF0000",
                       "azure" -> "#F0FFFF",
                       "peru" -> "#CD853F")

      val nums: Map[Int, Int] = Map()

      println( "colors 中的键为 : " + colors.keys )
      println( "colors 中的值为 : " + colors.values )
      println( "检测 colors 是否为空 : " + colors.isEmpty )
      println( "检测 nums 是否为空 : " + nums.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
colors 中的键为 : Set(red, azure, peru)
colors 中的值为 : MapLike(#FF0000, #F0FFFF, #CD853F)
检测 colors 是否为空 : false
检测 nums 是否为空 : true
```

Map 合并

你可以使用 **++** 运算符或 **Map.++()** 方法来连接两个 Map，Map 合并时会移除重复的 key。以下演示了两个 Map 合并的实例:

```
object Test {
   def main(args: Array[String]) {
      val colors1 = Map("red" -> "#FF0000",
                        "azure" -> "#F0FFFF",
                        "peru" -> "#CD853F")
      val colors2 = Map("blue" -> "#0033FF",
                        "yellow" -> "#FFFF00",
                        "red" -> "#FF0000")

      //  ++ 作为运算符
      var colors = colors1 ++ colors2
      println( "colors1 ++ colors2 : " + colors )

      //  ++ 作为方法
      colors = colors1.++(colors2)
      println( "colors1.++(colors2)) : " + colors )

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
colors1 ++ colors2 : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow -> #F
colors1.++(colors2)) : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow ->
```

输出 Map 的 keys 和 values

以下通过 foreach 循环输出 Map 中的 keys 和 values：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "http://www.runoob.com",
                      "baidu" -> "http://www.baidu.com",
                      "taobao" -> "http://www.taobao.com")

      sites.keys.foreach{ i =>
                          print( "Key = " + i )
                          println(" Value = " + sites(i) )}
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Key = runoob Value = http://www.runoob.com
Key = baidu Value = http://www.baidu.com
Key = taobao Value = http://www.taobao.com
```

查看 Map 中是否存在指定的 Key

你可以使用 **Map.contains** 方法来查看 Map 中是否存在指定的 Key。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "http://www.runoob.com",
                       "baidu" -> "http://www.baidu.com",
                       "taobao" -> "http://www.taobao.com")

      if( sites.contains( "runoob" )){
           println("runoob 键存在，对应的值为 :"  + sites("runoob"))
      }else{
           println("runoob 键不存在")
      }
      if( sites.contains( "baidu" )){
           println("baidu 键存在，对应的值为 :"  + sites("baidu"))
      }else{
           println("baidu 键不存在")
      }
      if( sites.contains( "google" )){
           println("google 键存在，对应的值为 :"  + sites("google"))
      }else{
           println("google 键不存在")
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
runoob 键存在，对应的值为 :http://www.runoob.com
baidu 键存在，对应的值为 :http://www.baidu.com
google 键不存在
```

Scala Map 方法

下表列出了 Scala Map 常用的方法：

方法	描述
def ++(xs: Map[(A, B)]): Map[A, B]	返回一个新的 Map，新的 Map xs 组成
def -(elem1: A, elem2: A, elems: A*): Map[A, B]	返回一个新的 Map, 移除 key 为 elem1, elem2 或其他 elems。
def --(xs: GTO[A]): Map[A, B]	返回一个新的 Map, 移除 xs 对象中对应的 key
def get(key: A): Option[B]	返回指定 key 的值
def iterator: Iterator[(A, B)]	创建新的迭代器，并输出 key/value 对
def addString(b: StringBuilder): StringBuilder	将 Map 中的所有元素附加到StringBuilder，可加入分隔符
def addString(b: StringBuilder, sep: String): StringBuilder	将 Map 中的所有元素附加到StringBuilder，可加入分隔符
def apply(key: A): B	返回指定键的值，如果不存在返回 Map 的默认方法

def clear(): Unit	清空 Map
def clone(): Map[A, B]	从一个 Map 复制到另一个 Map
def contains(key: A): Boolean	如果 Map 中存在指定 key，返回 true，否则返回 false。
def copyToArray(xs: Array[(A, B)]): Unit	复制集合到数组
def count(p: ((A, B)) => Boolean): Int	计算满足指定条件的集合元素数量
def default(key: A): B	定义 Map 的默认值，在 key 不存在时返回。
def drop(n: Int): Map[A, B]	返回丢弃前n个元素新集合
def dropRight(n: Int): Map[A, B]	返回丢弃最后n个元素新集合
def dropWhile(p: ((A, B)) => Boolean): Map[A, B]	从左向右丢弃元素，直到条件p不成立
def empty: Map[A, B]	返回相同类型的空 Map
def equals(that: Any): Boolean	如果两个 Map 相等(key/value 均相等)，返回true，否则返回false
def exists(p: ((A, B)) => Boolean): Boolean	判断集合中指定条件的元素是否存在
def filter(p: ((A, B))=> Boolean): Map[A, B]	返回满足指定条件的所有集合
def filterKeys(p: (A) => Boolean): Map[A, B]	返回符合指定条件的的不可变 Map
def find(p: ((A, B)) => Boolean): Option[(A, B)]	查找集合中满足指定条件的第一个元素
def foreach(f: ((A, B)) => Unit): Unit	将函数应用到集合的所有元素
def init: Map[A, B]	返回所有元素，除了最后一个
def isEmpty: Boolean	检测 Map 是否为空
def keys: Iterable[A]	返回所有的key/p>
def last: (A, B)	返回最后一个元素
def max: (A, B)	查找最大元素
def min: (A, B)	查找最小元素
def mkString: String	集合所有元素作为字符串显示
def product: (A, B)	返回集合中数字元素的积。
def remove(key: A): Option[B]	移除指定 key
def retain(p: (A, B) => Boolean): Map.this.type	如果符合满足条件的返回 true

def size: Int	返回 Map 元素的个数
def sum: (A, B)	返回集合中所有数字元素之和
def tail: Map[A, B]	返回一个集合中除了第一元素之外的其他元素
def take(n: Int): Map[A, B]	返回前 n 个元素
def takeRight(n: Int): Map[A, B]	返回后 n 个元素
def takeWhile(p: ((A, B)) => Boolean): Map[A, B]	返回满足指定条件的元素
def toArray: Array[(A, B)]	集合转数组
def toBuffer[B >: A]: Buffer[B]	返回缓冲区，包含了 Map 的所有元素
def toList: List[A]	返回 List，包含了 Map 的所有元素
def toSeq: Seq[A]	返回 Seq，包含了 Map 的所有元素
def toSet: Set[A]	返回 Set，包含了 Map 的所有元素
def toString(): String	返回字符串对象

更多方法可以参考 API文档

Scala 元组

与列表一样，元组也是不可变的，但与列表不同的是元组可以包含不同类型的元素。

元组的值是通过将单个的值包含在圆括号中构成的。例如：

```
val t = (1, 3.14, "Fred")
```

以上实例在元组中定义了三个元素，对应的类型分别为[Int, Double, java.lang.String]。

此外我们也可以使用以上方式来定义：

```
val t = new Tuple3(1, 3.14, "Fred")
```

元组的实际类型取决于它的元素的类型，比如 (99, "runoob") 是 Tuple2[Int, String]。 ('u', 'r', "the", 1, 4, "me") 为 Tuple6[Char, Char, String, Int, Int, String]。

目前 Scala 支持的元组最大长度为 22。对于更大长度你可以使用集合，或者扩展元组。

访问元组的元素可以通过数字索引，如下一个元组：

```
val t = (4,3,2,1)
```

我们可以使用 t._1 访问第一个元素， t._2 访问第二个元素，如下所示：

```
object Test {
   def main(args: Array[String]) {
      val t = (4,3,2,1)

      val sum = t._1 + t._2 + t._3 + t._4

      println( "元素之和为: " + sum )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
元素之和为: 10
```

迭代元组

你可以使用 **Tuple.productIterator()** 方法来迭代输出元组的所有元素：

```
object Test {
   def main(args: Array[String]) {
      val t = (4,3,2,1)

      t.productIterator.foreach{ i =>println("Value = " + i )}
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Value = 4
Value = 3
Value = 2
Value = 1
```

元组转为字符串

你可以使用 **Tuple.toString()** 方法将元组的所有元素组合成一个字符串，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val t = new Tuple3(1, "hello", Console)

      println("连接后的字符串为: " + t.toString() )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
连接后的字符串为: (1,hello,scala.Console$@4dd8dc3)
```

元素交换

你可以使用 **Tuple.swap** 方法来交换元组的元素。如下实例：

```
object Test {
   def main(args: Array[String]) {
      val t = new Tuple2("www.google.com", "www.runoob.com")

      println("交换后的元组: " + t.swap )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
交换后的元组: (www.runoob.com,www.google.com)
```

Scala Option(选项)

Scala Option(选项)类型用来表示一个值是可选的（有值或无值)。

Option[T] 是一个类型为 T 的可选值的容器： 如果值存在， Option[T] 就是一个 Some[T] ，如果不存在， Option[T] 就是对象 None 。

接下来我们来看一段代码：

```
// 虽然 Scala 可以不定义变量的类型，不过为了清楚些，我还是
// 把他显示的定义上了

val myMap: Map[String, String] = Map("key1" -> "value")
val value1: Option[String] = myMap.get("key1")
val value2: Option[String] = myMap.get("key2")

println(value1) // Some("value1")
println(value2) // None
```

在上面的代码中，myMap 一个是一个 Key 的类型是 String，Value 的类型是 String 的 hash map，但不一样的是他的 get() 返回的是一个叫 Option[String] 的类别。

Scala 使用 Option[String] 来告诉你：「我会想办法回传一个 String，但也可能没有 String 给你」。

myMap 里并没有 key2 这笔数据，get() 方法返回 None。

Option 有两个子类别，一个是 Some，一个是 None，当他回传 Some 的时候，代表这个函式成功地给了你一个 String，而你可以透过 get() 这个函式拿到那个 String，如果他返回的是 None，则代表没有字符串可以给你。

另一个实例：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "www.runoob.com", "google" -> "www.google.com")

      println("sites.get( \"runoob\" ) : " +  sites.get( "runoob" )) // Some(www.runoob.c
      println("sites.get( \"baidu\" ) : " +  sites.get( "baidu" ))  //  None
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
sites.get( "runoob" ) : Some(www.runoob.com)
sites.get( "baidu" ) : None
```

你也可以通过模式匹配来输出匹配值。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "www.runoob.com", "google" -> "www.google.com")

      println("show(sites.get( \"runoob\")) : " +
                                          show(sites.get( "runoob")) )
      println("show(sites.get( \"baidu\")) : " +
                                          show(sites.get( "baidu")) )
   }

   def show(x: Option[String]) = x match {
      case Some(s) => s
      case None => "?"
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
show(sites.get( "runoob")) : www.runoob.com
show(sites.get( "baidu")) : ?
```

getOrElse() 方法

你可以使用 getOrElse() 方法来获取元组中存在的元素或者使用其默认的值，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val a:Option[Int] = Some(5)
      val b:Option[Int] = None

      println("a.getOrElse(0): " + a.getOrElse(0) )
      println("b.getOrElse(10): " + b.getOrElse(10) )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a.getOrElse(0): 5
b.getOrElse(10): 10
```

isEmpty() 方法

你可以使用 isEmpty() 方法来检测元组中的元素是否为 None，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val a:Option[Int] = Some(5)
      val b:Option[Int] = None

      println("a.isEmpty: " + a.isEmpty )
      println("b.isEmpty: " + b.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a.isEmpty: false
b.isEmpty: true
```

Scala Option 常用方法

下表列出了 Scala Option 常用的方法：

方法	描述
def get: A	获取可选值
def isEmpty: Boolean	检测可选类型值是否为 None，是的话返回 true，否则返回 false
def productArity: Int	返回元素个数， A(x_1, ..., x_k), 返回 k
def productElement(n: Int): Any	获取指定的可选项，以 0 为起始。即 A(x1, ..., x_k), 返回 x(n+1)， 0 < n < k.
def exists(p: (A) => Boolean): Boolean	如果可选项中指定条件的元素是否存在且不为 None 返回 true，否则返回 false。
def filter(p: (A) => Boolean): Option[A]	如果选项包含有值，而且传递给 filter 的条件函数返回 true， filter 会返回 Some 实例。 否则，返回值为 None 。
def filterNot(p: (A) => Boolean): Option[A]	如果选项包含有值，而且传递给 filter 的条件函数返回 false， filter 会返回 Some 实例。 否则，返回值为 None 。
def flatMap[B](f: (A) | => Option[B]): Option[B]	如果选项包含有值，则传递给函数 f 处理后返回，否则返回 None
def foreach[U](f: (A) | => U): Unit	如果选项包含有值，则将每个值传递给函数 f， 否则不处理。
def getOrElse[B >: A](default: => B) |: B	如果选项包含有值，返回选项值，否则返回设定的默认值。
def isDefined: Boolean	如果可选值是 Some 的实例返回 true，否则返回 false。
def iterator: Iterator[A]	如果选项包含有值，迭代出可选值。如果可选值为空则返回空迭代器。
def map[B](f: (A) | => B): Option[B]	如果选项包含有值， 返回由函数 f 处理后的 Some，否则返回 None
def orElse[B >: A](alternative: => Option[B]) |: Option[B]	如果一个 Option 是 None ， orElse 方法会返回传名参数的值，否则，就直接返回这个 Option。
def orNull	如果选项包含有值返回选项值，否则返回 null。

Scala Iterator（迭代器）

Scala Iterator（迭代器）不是一个集合，它是一种用于访问集合的方法。

迭代器 it 的两个基本操作是 **next** 和 **hasNext** 。

调用 **it.next()** 会返回迭代器的下一个元素，并且更新迭代器的状态。

调用 **it.hasNext()** 用于检测集合中是否还有元素。

让迭代器 it 逐个返回所有元素最简单的方法是使用 while 循环：

```
object Test {
   def main(args: Array[String]) {
      val it = Iterator("Baidu", "Google", "Runoob", "Taobao")

      while (it.hasNext){
         println(it.next())
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Baidu
Google
Runoob
Taobao
```

查找最大与最小元素

你可以使用 **it.min** 和 **it.max** 方法从迭代器中查找最大与最小元素，实例如下:

```
object Test {
   def main(args: Array[String]) {
      val ita = Iterator(20,40,2,50,69, 90)
      val itb = Iterator(20,40,2,50,69, 90)

      println("最大元素是：" + ita.max )
      println("最小元素是：" + itb.min )

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
最大元素是：90
最小元素是：2
```

获取迭代器的长度

你可以使用 **it.size** 或 **it.length** 方法来查看迭代器中的元素个数。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val ita = Iterator(20,40,2,50,69, 90)
      val itb = Iterator(20,40,2,50,69, 90)

      println("ita.size 的值: " + ita.size )
      println("itb.length 的值: " + itb.length )

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
ita.size 的值: 6
itb.length 的值: 6
```

Scala Iterator 常用方法

下表列出了 Scala Iterator 常用的方法：

方法	描述
def hasNext: Boolean	如果还有可返回的元素，返回true。
def next(): A	返回迭代器的下一个元素，并且更新迭代器的状态
def ++(that: => Iterator[A]): Iterator[A]	合并两个迭代器
def ++[B >: A](that :=> GenTraversableOnce[B]) |: Iterator[B]	合并两个迭代器
def addString(b: StringBuilder): StringBuilder	添加一个字符串到 StringBuilder b
def addString(b: StringBuilder, sep: String): StringBuilder	添加一个字符串到 StringBuilder b，并指定分隔符

def buffered: BufferedIterator[A]	迭代器都转换成 BufferedIterator
def contains(elem: Any): Boolean	检测迭代器中是否包含指定元素
def copyToArray(xs: Array[A], start: Int, len: Int): Unit	将迭代器中选定的值传给数组
def count(p: (A) => Boolean): Int	返回迭代器元素中满足条件p的元素总数。
def drop(n: Int): Iterator[A]	返回丢弃前n个元素新集合
def dropWhile(p: (A) => Boolean): Iterator[A]	从左向右丢弃元素，直到条件p不成立
def duplicate: (Iterator[A], Iterator[A])	生成两个能分别返回迭代器所有元素的迭代器。
def exists(p: (A) => Boolean): Boolean	返回一个布尔值，指明迭代器元素中是否存在满足p的元素。
def filter(p: (A) => Boolean): Iterator[A]	返回一个新迭代器 ，指向迭代器元素中所有满足条件p的元素。
def filterNot(p: (A) => Boolean): Iterator[A]	返回一个迭代器，指向迭代器元素中不满足条件p的元素。
def find(p: (A) => Boolean): Option[A]	返回第一个满足p的元素或None。注意：如果找到满足条件的元素，迭代器会被置于该元素之后；如果没有找到，会被置于终点。
def flatMap[B](f: (A) | => GenTraversableOnce[B]): Iterator[B]	针对迭代器的序列中的每个元素应用函数f，并返回指向结果序列的迭代器。
def forall(p: (A) => Boolean): Boolean	返回一个布尔值，指明 it 所指元素是否都满足p。
def foreach(f: (A) => Unit): Unit	在迭代器返回的每个元素上执行指定的程序 f
def hasDefiniteSize: Boolean	如果迭代器的元素个数有限则返回true（缺省等同于isEmpty）
def indexOf(elem: B): Int	返回迭代器的元素中index等于x的第一个元素。注意：迭代器会越过这个元素。
def indexWhere(p: (A) => Boolean): Int	返回迭代器的元素中下标满足条件p的元素。注意：迭代器会越过这个元素。
def isEmpty: Boolean	检查it是否为空, 为空返回 true，否则返回false（与hasNext相反）。
def isTraversableAgain: Boolean	Tests whether this Iterator can be repeatedly traversed.

def length: Int	返回迭代器元素的数量。
def map[B](f: (A) | => B): Iterator[B]	将 it 中的每个元素传入函数 f 后的结果生成新的迭代器。
def max: A	返回迭代器迭代器元素中最大的元素。
def min: A	返回迭代器迭代器元素中最小的元素。
def mkString: String	将迭代器所有元素转换成字符串。
def mkString(sep: String): String	将迭代器所有元素转换成字符串，并指定分隔符。
def nonEmpty: Boolean	检查容器中是否包含元素（相当于 hasNext）。
def padTo(len: Int, elem: A): Iterator[A]	首先返回迭代器所有元素，追加拷贝 elem 直到长度达到 len。
def patch(from: Int, patchElems: Iterator[B], replaced: Int): Iterator[B]	返回一个新迭代器，其中自第 from 个元素开始的 replaced 个元素被迭代器所指元素替换。
def product: A	返回迭代器所指数值型元素的积。
def sameElements(that: Iterator[_]): Boolean	判断迭代器和指定的迭代器参数是否依次返回相同元素
def seq: Iterator[A]	返回集合的系列视图
def size: Int	返回迭代器的元素数量
def slice(from: Int, until: Int): Iterator[A]	返回一个新的迭代器，指向迭代器所指向的序列中从开始于第 from 个元素、结束于第 until 个元素的片段。
def sum: A	返回迭代器所指数值型元素的和
def take(n: Int): Iterator[A]	返回前 n 个元素的新迭代器。
def toArray: Array[A]	将迭代器指向的所有元素归入数组并返回。
def toBuffer: Buffer[B]	将迭代器指向的所有元素拷贝至缓冲区 Buffer。
def toIterable: Iterable[A]	Returns an Iterable containing all elements of this traversable or iterator. This will not terminate for infinite iterators.
def toIterator: Iterator[A]	把迭代器的所有元素归入一个Iterator容器并返回。
def toList: List[A]	把迭代器的所有元素归入列表并返回
def toMap[T, U]: Map[T, U]	将迭代器的所有键值对归入一个Map并返回。
def toSeq: Seq[A]	将代器的所有元素归入一个Seq容器并返回。
def toString(): String	将迭代器转换为字符串
def zip[B](that: Iterator[B]) |: Iterator[(A, B)]	返回一个新迭代器，指向分别由迭代器和指定的迭代器 that 元素一一对应而成的二元组序列

更多方法可以参考 API文档

Scala 类和对象

类是对象的抽象，而对象是类的具体实例。类是抽象的，不占用内存，而对象是具体的，占用存储空间。类是用于创建对象的蓝图，它是一个定义包括在特定类型的对象中的方法和变量的软件模板。

我们可以使用 new 关键字来创建类的对象，实例如下：

```
class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点: " + x);
      println ("y 的坐标点: " + y);
   }
}
```

Scala中的类不声明为public，一个Scala源文件中可以有多个类。

以上实例的类定义了两个变量 **x** 和 **y** ，一个方法：**move**，方法没有返回值。

Scala 的类定义可以有参数，称为类参数，如上面的 xc, yc，类参数在整个类中都可以访问。

接着我们可以使用 new 来实例化类，并访问类中的方法和变量：

```
import java.io._

class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点: " + x);
      println ("y 的坐标点: " + y);
   }
}

object Test {
   def main(args: Array[String]) {
      val pt = new Point(10, 20);

      // 移到一个新的位置
      pt.move(10, 10);
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 的坐标点: 20
y 的坐标点: 30
```

Scala 继承

Scala继承一个基类跟Java很相似，只多了两点限制：

- 1、重写方法需要override关键字
- 2、只有主构造函数才可以往基类的构造函数里写参数。

Scala 的副构造函数必须调用主构造函数或另一个构造函数，在 Scala 里主构造函数如同一道关卡，类的实例需要通过他来初始化。

接下来让我们来看哥实例：

```
class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点: " + x);
      println ("y 的坐标点: " + y);
   }
}

class Location(override val xc: Int, override val yc: Int,
   val zc :Int) extends Point(xc, yc){
   var z: Int = zc

   def move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
      println ("x 的坐标点 : " + x);
      println ("y 的坐标点 : " + y);
      println ("z 的坐标点 : " + z);
   }
}
```

Scala 使用 extends 关键字来继承一个类。实例中 Location 类继承了 Point 类。Point 称为父类(基类)，Location 称为子类。

继承会继承父类的所有属性和方法，Scala 只允许继承一个父类。

实例如下：

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc
   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点 : " + x);
      println ("y 的坐标点 : " + y);
   }
}

class Location(override val xc: Int, override val yc: Int,
   val zc :Int) extends Point(xc, yc){
   var z: Int = zc

   def move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
      println ("x 的坐标点 : " + x);
      println ("y 的坐标点 : " + y);
      println ("z 的坐标点 : " + z);
   }
}

object Test {
   def main(args: Array[String]) {
      val loc = new Location(10, 20, 15);

      // 移到一个新的位置
      loc.move(10, 10, 5);
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 的坐标点 : 20
y 的坐标点 : 30
z 的坐标点 : 20
```

Scala 单例对象

在 Scala 中，是没有 static 这个东西的，但是它也为我们提供了单例模式的实现方法，那就是使用关键字 object。

Scala 中使用单例模式时，除了定义的类之外，还要定义一个同名的 object 对象，它和类的区别是，object对象不能带参数。

当单例对象与某个类共享同一个名称时，他被称作是这个类的伴生对象：companion object。你必须在同一个源文件里定义类和它的伴生对象。类被称为是这个单例对象的伴生类：companion class。类和它的伴生对象可以互相访问其私有成员。

单例对象实例

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc
   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
   }
}

object Test {
   def main(args: Array[String]) {
      val point = new Point(10, 20)
      printPoint

      def printPoint{
         println ("x 的坐标点 : " + point.x);
         println ("y 的坐标点 : " + point.y);
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 的坐标点 : 10
y 的坐标点 : 20
```

伴生对象实例

```
/* 文件名：Marker.scala
 * author:菜鸟教程
 * url:www.runoob.com
 */

// 私有构造方法
class Marker private(val color:String) {

  println("创建" + this)

  override def toString(): String = "颜色标记："+ color

}

// 伴生对象，与类共享名字，可以访问类的私有属性和方法
object Marker{

    private val markers: Map[String, Marker] = Map(
      "red" -> new Marker("red"),
      "blue" -> new Marker("blue"),
      "green" -> new Marker("green")
    )

    def apply(color:String) = {
      if(markers.contains(color)) markers(color) else null
    }

    def getMarker(color:String) = {
      if(markers.contains(color)) markers(color) else null
    }
    def main(args: Array[String]) {
        println(Marker("red"))
        // 单例函数调用，省略了.(点)符号
        println(Marker getMarker "blue")
    }
}
```

执行以上代码，输出结果为：

```
$ scalac Marker.scala
$ scala Marker
创建颜色标记：red
创建颜色标记：blue
创建颜色标记：green
颜色标记：red
颜色标记：blue
```

Scala Trait(特征)

Scala Trait(特征) 相当于 Java 的接口，实际上它比接口还功能强大。

与接口不同的是，它还可以定义属性和方法的实现。

一般情况下Scala的类只能够继承单一父类，但是如果是 Trait(特征) 的话就可以继承多个，从结果来看就是实现了多重继承。

Trait(特征) 定义的方式与类类似，但它使用的关键字是 **trait**，如下所示：

```
trait Equal {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

以上Trait(特征)由两个方法组成：**isEqual** 和 **isNotEqual**。isEqual 方法没有定义方法的实现，isNotEqual定义了方法的实现。子类继承特征可以实现未被实现的方法。所以其实 Scala Trait(特征)更像 Java 的抽象类。

以下演示了特征的完整实例：

```
/* 文件名：Test.scala
 * author:菜鸟教程
 * url:www.runoob.com
 */
trait Equal {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)
}

class Point(xc: Int, yc: Int) extends Equal {
  var x: Int = xc
  var y: Int = yc
  def isEqual(obj: Any) =
    obj.isInstanceOf[Point] &&
    obj.asInstanceOf[Point].x == x
}

object Test {
   def main(args: Array[String]) {
      val p1 = new Point(2, 3)
      val p2 = new Point(2, 4)
      val p3 = new Point(3, 3)

      println(p1.isNotEqual(p2))
      println(p1.isNotEqual(p3))
      println(p1.isNotEqual(2))
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
false
true
true
```

特征构造顺序

特征也可以有构造器，由字段的初始化和其他特征体中的语句构成。这些语句在任何混入该特征的对象在构造是都会被执行。

构造器的执行顺序：

- 调用超类的构造器；
- 特征构造器在超类构造器之后、类构造器之前执行；
- 特质由左到右被构造；
- 每个特征当中，父特质先被构造；
- 如果多个特征共有一个父特质，父特质不会被重复构造
- 所有特征被构造完毕，子类被构造。

构造器的顺序是类的线性化的反向。线性化是描述某个类型的所有超类型的一种技术规格。

Scala 模式匹配

Scala 提供了强大的模式匹配机制，应用也非常广泛。

一个模式匹配包含了一系列备选项，每个都开始于关键字 **case**。每个备选项都包含了一个模式及一到多个表达式。箭头符号 **=>** 隔开了模式和表达式。

以下是一个简单的整型值模式匹配实例：

```
object Test {
   def main(args: Array[String]) {
      println(matchTest(3))

   }
   def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
many
```

match 对应 Java 里的 switch，但是写在选择器表达式之后。即： 选择器 **match {备选项}**。

match 表达式通过以代码编写的先后次序尝试每个模式来完成计算，只要发现有一个匹配的case，剩下的case不会继续匹配。

接下来我们来看一个不同数据类型的模式匹配：

```
object Test {
   def main(args: Array[String]) {
      println(matchTest("two"))
      println(matchTest("test"))
      println(matchTest(1))
      println(matchTest(6))

   }
   def matchTest(x: Any): Any = x match {
      case 1 => "one"
      case "two" => 2
      case y: Int => "scala.Int"
      case _ => "many"
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
2
many
one
scala.Int
```

实例中第一个 case 对应整型数值 1，第二个 case 对应字符串值 two，第二个 case 对应字符串值 two，第三个 case 对应类型模式，用于判断传入的值是否为整型，相比使用isInstanceOf来判断类型，使用模式匹配更好。第四个 case 表示默认的全匹配备选项，即没有找到其他匹配时的匹配项，类似 switch 中的 default。

使用样例类

使用了case关键字的类定义就是就是样例类(case classes)，样例类是种特殊的类，经过优化以用于模式匹配。

以下是样例类的简单实例:

```
object Test {
   def main(args: Array[String]) {
       val alice = new Person("Alice", 25)
    val bob = new Person("Bob", 32)
       val charlie = new Person("Charlie", 32)

    for (person <- List(alice, bob, charlie)) {
        person match {
            case Person("Alice", 25) => println("Hi Alice!")
            case Person("Bob", 32) => println("Hi Bob!")
            case Person(name, age) =>
               println("Age: " + age + " year, name: " + name + "?")
         }
      }
   }
   // 样例类
   case class Person(name: String, age: Int)
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Hi Alice!
Hi Bob!
Age: 32 year, name: Charlie?
```

在声明样例类时，下面的过程自动发生了：

- 构造器的每个参数都成为val，除非显式被声明为var，但是并不推荐这么做；
- 在伴生对象中提供了apply方法，所以可以不使用new关键字就可构建对象；
- 提供unapply方法使模式匹配可以工作；
- 生成toString、equals、hashCode和copy方法，除非显示给出这些方法的定义。

Scala 正则表达式

Scala 通过 scala.util.matching 包种的 **Regex** 类来支持正则表达式。以下实例演示了使用正则表达式查找单词 **Scala** :

```
import scala.util.matching.Regex

object Test {
   def main(args: Array[String]) {
      val pattern = "Scala".r
      val str = "Scala is Scalable and cool"

      println(pattern findFirstIn str)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Some(Scala)
```

实例中使用 String 类的 r() 方法构造了一个Regex对象。

然后使用 findFirstIn 方法找到首个匹配项。

如果需要查看所有的匹配项可以使用 findAllIn 方法。

你可以使用 mkString( ) 方法来连接正则表达式匹配结果的字符串，并可以使用管道(|)来设置不同的模式：

```
import scala.util.matching.Regex

object Test {
   def main(args: Array[String]) {
      val pattern = new Regex("(S|s)cala")  // 首字母可以是大写 S 或小写 s
      val str = "Scala is scalable and cool"

      println((pattern findAllIn str).mkString(","))   // 使用逗号 , 连接返回结果
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Scala,scala
```

如果你需要将匹配的文本替换为指定的关键词，可以使用 **replaceFirstIn( )** 方法来替换第一个匹配项，使用 **replaceAllIn( )** 方法替换所有匹配项，实例如下:

```
object Test {
   def main(args: Array[String]) {
      val pattern = "(S|s)cala".r
      val str = "Scala is scalable and cool"

      println(pattern replaceFirstIn(str, "Java"))
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Java is scalable and cool
```

正则表达式

Scala 的正则表达式继承了 Java 的语法规则，Java 则大部分使用了 Perl 语言的规则。

下表我们给出了常用的一些正则表达式规则：

表达式	匹配规则
^	匹配输入字符串开始的位置。
$	匹配输入字符串结尾的位置。
.	匹配除"\r\n"之外的任何单个字符。
[...]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^...]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。
\A	匹配输入字符串开始的位置（无多行支持）
\z	字符串结尾(类似$，但不受处理多行选项的影响)
\Z	字符串结尾或行尾(不受处理多行选项的影响)
re*	重复零次或更多次
re+	重复一次或更多次
re?	重复零次或一次
re{ n}	重复n次
re{ n,}	
re{ n, m}	重复n到m次
a|b	匹配 a 或者 b
(re)	匹配 re,并捕获文本到自动命名的组里

(?: re)	匹配 re,不捕获匹配的文本，也不给此分组分配组号
(?> re)	贪婪子表达式
\w	匹配字母或数字或下划线或汉字
\W	匹配任意不是字母，数字，下划线，汉字的字符
\s	匹配任意的空白符,相等于 [\t\n\r\f]
\S	匹配任意不是空白符的字符
\d	匹配数字，类似 [0-9]
\D	匹配任意非数字的字符
\G	当前搜索的开头
\n	换行符
\b	通常是单词分界位置，但如果在字符类里使用代表退格
\B	匹配不是单词开头或结束的位置
\t	制表符
\Q	开始引号：**\Q(a+b)*3\E** 可匹配文本 "(a+b)*3"。
\E	结束引号：**\Q(a+b)*3\E** 可匹配文本 "(a+b)*3"。

正则表达式实例

实例	描述
.	匹配除"\r\n"之外的任何单个字符。
[Rr]uby	匹配 "Ruby" 或 "ruby"
rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配小写字母 ：aeiou
[0-9]	匹配任何数字，类似 [0123456789]
[a-z]	匹配任何 ASCII 小写字母
[A-Z]	匹配任何 ASCII 大写字母
[a-zA-Z0-9]	匹配数字，大小写字母
[^aeiou]	匹配除了 aeiou 其他字符
[^0-9]	匹配除了数字的其他字符
\d	匹配数字，类似: [0-9]
\D	匹配非数字，类似: [^0-9]
\s	匹配空格，类似: [\t\r\n\f]
\S	匹配非空格，类似: [^ \t\r\n\f]
\w	匹配字母，数字，下划线，类似: [A-Za-z0-9_]
\W	匹配非字母，数字，下划线，类似: [^A-Za-z0-9_]
ruby?	匹配 "rub" 或 "ruby": y 是可选的
ruby*	匹配 "rub" 加上 0 个或多个的 y。
ruby+	匹配 "rub" 加上 1 个或多个的 y。
\d{3}	刚好匹配 3 个数字。
\d{3,}	匹配 3 个或多个数字。
\d{3,5}	匹配 3 个、4 个或 5 个数字。
\D\d+	无分组： + 重复 \d
(\D\d)+/	分组： + 重复 \D\d 对
([Rr]uby(,)?)+	匹配 "Ruby"、"Ruby, ruby, ruby"，等等

注意上表中的每个字符使用了两个反斜线。这是因为在 Java 和 Scala 中字符串中的反斜线是转义字符。所以如果你要输出 ..，你需要在字符串中写成 .\. 来获取一个反斜线。查看以下实例：

```
import scala.util.matching.Regex

object Test {
   def main(args: Array[String]) {
      val pattern = new Regex("abl[ae]\\d+")
      val str = "ablaw is able1 and cool"

      println((pattern findAllIn str).mkString(","))
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
able1
```

Scala 异常处理

Scala 的异常处理和其它语言比如 Java 类似。

Scala 的方法可以通过抛出异常的方法的方式来终止相关代码的运行，不必通过返回值。

抛出异常

Scala 抛出异常的方法和 Java一样，使用 throw 方法，例如，抛出一个新的参数异常：

```
throw new IllegalArgumentException
```

捕获异常

异常捕捉的机制与其他语言中一样，如果有异常发生，catch字句是按次序捕捉的。因此，在catch字句中，越具体的异常越要靠前，越普遍的异常越靠后。 如果抛出的异常不在catch字句中，该异常则无法处理，会被升级到调用者处。

捕捉异常的catch子句，语法与其他语言中不太一样。在Scala里，借用了模式匹配的思想来做异常的匹配，因此，在catch的代码里，是一系列case字句，如下例所示：

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
   def main(args: Array[String]) {
      try {
         val f = new FileReader("input.txt")
      } catch {
         case ex: FileNotFoundException =>{
            println("Missing file exception")
         }
         case ex: IOException => {
            println("IO Exception")
         }
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Missing file exception
```

catch字句里的内容跟match里的case是完全一样的。由于异常捕捉是按次序，如果最普遍的异常，Throwable，写在最前面，则在它后面的case都捕捉不到，因此需要将它写在最后面。

finally 语句

finally 语句用于执行不管是正常处理还是有异常发生时都需要执行的步骤，实例如下：

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
   def main(args: Array[String]) {
      try {
         val f = new FileReader("input.txt")
      } catch {
         case ex: FileNotFoundException => {
            println("Missing file exception")
         }
         case ex: IOException => {
            println("IO Exception")
         }
      } finally {
         println("Exiting finally...")
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Missing file exception
Exiting finally...
```

Scala 提取器(Extractor)

提取器是从传递给它的对象中提取出构造该对象的参数。

Scala 标准库包含了一些预定义的提取器，我们会大致的了解一下它们。

Scala 提取器是一个带有unapply方法的对象。unapply方法算是apply方法的反向操作：unapply接受一个对象，然后从对象中提取值，提取的值通常是用来构造该对象的值。

以下实例演示了邮件地址的提取器对象：

```
object Test {
   def main(args: Array[String]) {

      println ("Apply 方法 : " + apply("Zara", "gmail.com"));
      println ("Unapply 方法 : " + unapply("Zara@gmail.com"));
      println ("Unapply 方法 : " + unapply("Zara Ali"));

   }
   // 注入方法 (可选)
   def apply(user: String, domain: String) = {
      user +"@"+ domain
   }

   // 提取方法（必选）
   def unapply(str: String): Option[(String, String)] = {
      val parts = str split "@"
      if (parts.length == 2){
         Some(parts(0), parts(1))
      }else{
         None
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Apply 方法 : Zara@gmail.com
Unapply 方法 : Some((Zara,gmail.com))
Unapply 方法 : None
```

以上对象定义了两个方法： **apply** 和 **unapply** 方法。通过 apply 方法我们无需使用 new 操作就可以创建对象。所以你可以通过语句 Test("Zara", "gmail.com") 来构造一个字符串 "Zara@gmail.com"。

unapply方法算是apply方法的反向操作：unapply接受一个对象，然后从对象中提取值，提取的值通常是用来构造该对象的值。实例中我们使用 Unapply 方法从对象中提取用户名和邮件地址的后缀。

实例中 unapply 方法在传入的字符串不是邮箱地址时返回 None。代码演示如下：

```
unapply("Zara@gmail.com") 相等于 Some("Zara", "gmail.com")
unapply("Zara Ali") 相等于 None
```

提取器使用模式匹配

在我们实例化一个类的时，可以带上0个或者多个的参数，编译器在实例化的时会调用 apply 方法。我们可以在类和对象中都定义 apply 方法。

就像我们之前提到过的，unapply 用于提取我们指定查找的值，它与 apply 的操作相反。 当我们在提取器对象中使用 match 语句是，unapply 将自动执行，如下所示：

```
object Test {
   def main(args: Array[String]) {

      val x = Test(5)
      println(x)

      x match
      {
         case Test(num) => println(x + " 是 " + num + " 的两倍！")
         //unapply 被调用
         case _ => println("无法计算")
      }

   }
   def apply(x: Int) = x*2
   def unapply(z: Int): Option[Int] = if (z%2==0) Some(z/2) else None
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
10
10 是 5 的两倍！
```

Scala 文件 I/O

Scala 进行文件写操作，直接用的都是 java中 的 I/O 类 （**java.io.File**)：

```
import java.io._

object Test {
   def main(args: Array[String]) {
      val writer = new PrintWriter(new File("test.txt" ))

      writer.write("菜鸟教程")
      writer.close()
   }
}
```

执行以上代码，会在你的当前目录下生产一个 test.txt 文件，文件内容为"菜鸟教程":

```
$ scalac Test.scala
$ scala Test
$ cat test.txt
菜鸟教程
```

从屏幕上读取用户输入

有时候我们需要接收用户在屏幕输入的指令来处理程序。实例如下：

```
object Test {
   def main(args: Array[String]) {
      print("请输入菜鸟教程官网 : " )
      val line = Console.readLine

      println("谢谢，你输入的是: " + line)
   }
}
```

执行以上代码，屏幕上会显示如下信息:

```
$ scalac Test.scala
$ scala Test
请输入菜鸟教程官网 : www.runoob.com
谢谢，你输入的是: www.runoob.com
```

从文件上读取内容

从文件读取内容非常简单。我们可以使用 Scala 的 **Source** 类及伴生对象来读取文件。以下实例演示了从 "test.txt"(之前已创建过) 文件中读取内容:

```
import scala.io.Source

object Test {
   def main(args: Array[String]) {
      println("文件内容为:" )

      Source.fromFile("test.txt" ).foreach{
         print
      }
   }
}
```

执行以上代码，输出结果为:

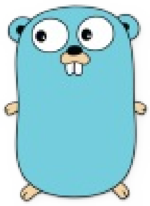
```
$ scalac Test.scala
$ scala Test
文件内容为:
菜鸟教程
```

W3School Go语言 教程

作者：W3School

来源：Go语言 教程

Go 语言简介



Go 是一个开源的编程语言，它能让构造简单、可靠且高效的软件变得容易。

Go是从2007年末由Robert Griesemer, Rob Pike, Ken Thompson主持开发，后来还加入了Ian Lance Taylor, Russ Cox等人，并最终于2009年11月开源，在2012年早些时候发布了Go 1稳定版本。现在Go的开发已经是完全开放的，并且拥有一个活跃的社区。

Go 语言特色

- 简洁、快速、安全
- 并行、有趣、开源
- 内存管理、v数组安全、编译迅速

Go 语言用途

Go 语言被设计成一门应用于搭载 Web 服务器，存储集群或类似用途的巨型中央服务器的系统编程语言。

对于高性能分布式系统领域而言，Go 语言无疑比大多数其它语言有着更高的开发效率。它提供了海量并行的支持，这对于游戏服务端的开发而言是再好不过了。

第一个 Go 程序

接下来我们来编写第一个 Go 程序 hello.go（Go 语言源文件的扩展是 .go），代码如下：

实例

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

运行实例 »

执行以上代码输出

```
$ go run hello.go Hello,  World!
```

Go 语言环境安装

Go 语言支持以下系统：

- Linux
- FreeBSD
- Mac OS X（也称为 Darwin）
- Window

安装包下载地址为：https://golang.org/dl/。

各个系统对应的包名：

操作系统	包名
Windows	go1.4.windows-amd64.msi
Linux	go1.4.linux-amd64.tar.gz
Mac	go1.4.darwin-amd64-osx10.8.pkg
FreeBSD	go1.4.freebsd-amd64.tar.gz

File name		Kind	OS	Arch	SHA1 Checksum
go1.4.2.src.tar.gz	源码包	**Source**			**460caac03379f746c473814a65223397e9c9a2f6**
go1.4.2.darwin-386-osx10.6.tar.gz		Archive	OS X 10.6+	32-bit	fb3e6b30f4e1b1be47bbb98d79dd53da8dec24ec
go1.4.2.darwin-386-osx10.8.tar.gz		Archive	OS X 10.8+	32-bit	65f5610fdb38febd869aeffbd426c83b650bb408
go1.4.2.darwin-386-osx10.6.pkg		Installer	OS X 10.6+	32-bit	3ed569ce33616d5d36f963e5d7cefb55727c8621
go1.4.2.darwin-386-osx10.8.pkg		Installer	OS X 10.8+	32-bit	7f3fb2438fa0212febef13749d8d144934bb1c80
go1.4.2.darwin-amd64-osx10.6.tar.gz		Archive	OS X 10.6+	64-bit	00c3f9a03daff818b2132ac31d57f054925c60e7
go1.4.2.darwin-amd64-osx10.8.tar.gz		Archive	OS X 10.8+	64-bit	58a04b3eb9853c75319d9076df6f3ac8b7430f7f
go1.4.2.darwin-amd64-osx10.6.pkg		Installer	OS X 10.6+	64-bit	3fa5455e211a70c0a920abd53cb3093269c5149c
go1.4.2.darwin-amd64-osx10.8.pkg		**Installer**	**OS X 10.8+**	**64-bit**	**8fde619d48864cb1c77ddc2a1aec0b7b20406b38**
go1.4.2.linux-386.tar.gz	MAC系统	Archive	Linux	32-bit	50557248e89b6e38d395fda93b2f96b2b860a26a
go1.4.2.linux-amd64.tar.gz		**Archive**	**Linux**	**64-bit**	**5020af94b52b65cc9b6f11d50a67e4bae07b0aff**
go1.4.2.windows-386.zip		Archive	Windows	32-bit	0e074e66a7816561d7947ff5c3514be96f347dc4
go1.4.2.windows-386.msi		Installer	Windows	32-bit	e8bd3d87cb52441b2c9aee7c2c5f5ce7ffccc832
go1.4.2.windows-amd64.zip	Window	Archive	Windows	64-bit	91b229a3ff0a1ce6e791c832b0b4670bfc5457b5
go1.4.2.windows-amd64.msi		**Installer**	**Windows**	**64-bit**	**a914f3dad5521a8f658dce3e1575f3b6792975f0**

UNIX/Linux/Mac OS X, 和 FreeBSD 安装

以下介绍了在UNIX/Linux/Mac OS X, 和 FreeBSD系统下使用源码安装方法：

1、下载源码包：go1.4.linux-amd64.tar.gz。

2、将下载的源码包解压至 /usr/local目录。

```
tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz
```

3、将 /usr/local/go/bin 目录添加至PATH环境变量：

```
export PATH=$PATH:/usr/local/go/bin
```

注意：MAC 系统下你可以使用 .pkg 结尾的安装包直接双击来完成安装，安装目录在 /usr/local/go/ 下。

Windows 系统下安装

Windows 下可以使用 .msi 后缀(在下载列表中可以找到该文件，如go1.4.2.windows-amd64.msi)的安装包来安装。

默认情况下.msi文件会安装在 c:\Go 目录下。你可以将 c:\Go\bin 目录添加到 PATH 环境变量中。添加后你需要重启命令窗口才能生效。

安装测试

创建工作目录 C:\>Go_WorkSpace。

文件名: test.go，代码如下：

```
package main

import "fmt"

func main() {
   fmt.Println("Hello, World!")
}
```

使用 go 命令执行以上代码输出结果如下：

```
C:\Go_WorkSpace>go run test.go

Hello, World!
```

Go 语言结构

在我们开始学习 GO 编程语言的基础构建模块前，让我们先来了解 Go 语言最简单程序的结构。

Go Hello World 实例

Go 语言的基础组成有以下几个部分：

- 包声明
- 引入包
- 函数
- 变量
- 语句 & 表达式
- 注释

接下来让我们来看下简单的代码，该代码输出了"Hello World!":

```
package main import  "fmt" func main()  {  /* 这是我的第一个简单的程序 */ fmt.Println("Hello,
```

让我们来看下以上程序的各个部分：

1. 第一行代码 *package main* 定义了包名。你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。
2. 下一行 *import "fmt"* 告诉 Go 编译器这个程序需要使用 fmt 包（的函数，或其他元素），fmt 包实现了格式化 IO（输入/输出）的函数。
3. 下一行 *func main()* 是程序开始执行的函数。main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（如果有 init() 函数则会先执行该函数）。
4. 下一行 /.../ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 / 开头，并以 / 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。
5. 下一行 *fmt.Println(...)* 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。 使用 fmt.Print("hello, world\n") 可以得到相同的结果。 Print 和 Println 这两个函数也支持使用变量，如：fmt.Println(arr)。如果没有特别指定，它们会以默认的打印格式将变量 arr 输出到控制台。

6. 当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 private ）。

执行 Go 程序

让我们来看下如何编写 Go 代码并执行它。步骤如下：

1. 打开编辑器如Sublime2，将以上代码添加到编辑器中。
2. 将以上代码保存为 *hello.go*
3. 打开命令行，并进入程序文件保存的目录中。
4. 输入命令 *go run hello.go* 并按回车执行代码。
5. 如果操作正确你将在屏幕上看到 *"Hello World!"* 字样的输出。

```
$ go run hello.go Hello,  World!
```

Go 语言基础语法

上一章节我们已经了解了 Go 语言的基本组成结构，本章节我们将学习 Go 语言的基础语法。

Go 标记

Go 程序可以由多个标记组成，可以是关键字，标识符，常量，字符串，符号。如以下 GO 语句由 6 个标记组成：

```
fmt.Println("Hello, World!")
```

6 个标记是(每行一个)：

```
1\. fmt
2\. .
3\. Println
4\. (
5\. "Hello, World!"
6\. )
```

行分隔符

在 Go 程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号 ; 结尾，因为这些工作都将由 Go 编译器自动完成。

如果你打算将多个语句写在同一行，它们则必须使用 ; 人为区分，但在实际开发中我们并不鼓励这种做法。

以下为两个语句：

```
fmt.Println("Hello, World!")
fmt.Println("w3cschool菜鸟教程：w3cschool.cc")
```

注释

注释不会被编译，每一个包应该有相关注释。

单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 / 开头，并以 / 结尾。如：

```
// 单行注释
/*
 Author by w3cschool菜鸟教程
 我是多行注释
 */
```

标识符

标识符用来命名变量、类型等程序实体。一个标识符实际上就是一个或是多个字母(A~Z和a~z)数字(0~9)、下划线_组成的序列，但是第一个字符必须是字母或下划线而不能是数字。

以下是有效的标识符：

```
mahesh   kumar   abc   move_name   a_123
myname50   _temp   j   a23b9   retVal
```

以下是无效的标识符：

- 1ab（以数字开头）
- case（Go 语言的关键字）
- a+b（运算符是不允许的）

关键字

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符：

append	bool	byte	cap	close	complex	complex64	complex
copy	false	float32	float64	imag	int	int8	int16
int32	int64	iota	len	make	new	nil	panic
print	println	real	recover	string	true	uint	uint8

程序一般由关键字、常量、变量、运算符、类型和函数组成。

程序中可能会使用到这些分隔符：括号 ()，中括号 [] 和大括号 {}。

程序中可能会使用到这些标点符号：.、,、;、: 和 …。

Go 语言的空格

Go 语言中变量的声明必须使用空格隔开，如：

```
var age int;
```

语句中适当使用空格能让程序看易阅读。

无空格：

```
fruit=apples+oranges;
```

在变量与运算符间加入空格，程序看起来更加美观，如：

```
fruit = apples + oranges;
```

Go 语言数据类型

在 Go 编程语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

Go 语言按类别有以下几种数据类型：

序号	类型和描述
布尔型	布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。
数字类型	整型 int 和浮点型 float，Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。
字符串类型:	字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。Go语言的字符串的字节使用UTF-8编码标识Unicode文本。
派生类型:	包括：(a) 指针类型（Pointer） (b) 数组类型 (c) 结构化类型(struct) (d) 联合体类型 (union) (e) 函数类型 (f) 切片类型 (g) 接口类型（interface） (h) Map 类型 (i) Channel 类型

数字类型

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

序号	类型和描述
uint8	无符号 8 位整型 (0 到 255)
uint16	无符号 16 位整型 (0 到 65535)
uint32	无符号 32 位整型 (0 到 4294967295)
uint64	无符号 64 位整型 (0 到 18446744073709551615)
int8	有符号 8 位整型 (-128 到 127)
int16	有符号 16 位整型 (-32768 到 32767)
int32	有符号 32 位整型 (-2147483648 到 2147483647)
int64	有符号 64 位整型 (-9223372036854775808 到 9223372036854775807)

浮点型：

序号	类型和描述
float32	IEEE-754 32位浮点型数
float64	IEEE-754 64位浮点型数
complex64	32 位实数和虚数
complex128	64 位实数和虚数

其他数字类型

以下列出了其他更多的数字类型：

序号	类型和描述
byte	类似 uint8
rune	类似 int32
uint	32 或 64 位
int	与 uint 一样大小
uintptr	无符号整型，用于存放一个指针

Go 语言变量

变量来源于数学，是计算机语言中能储存计算结果或能表示值抽象概念。变量可以通过变量名访问。

Go 语言变量名由字母、数字、下划线组成，其中首个字母不能为数字。

声明变量的一般形式是使用 var 关键字：

```
var identifier type
```

变量声明

第一种，指定变量类型，声明后若不赋值，使用默认值。

```
var v_name v_type
v_name = value
```

第二种，根据值自行判定变量类型。

```
var v_name = value
```

第三种，省略var, 注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误。

```
v_name := value

// 例如
var a int = 10
var b = 10
c : = 10
```

实例如下：

```
package main
var a = "w3cschool菜鸟教程"
var b string = "w3cschool.cc"
var c bool

func main(){
    println(a, b, c)
}
```

以上实例执行结果为：

```
w3cschool菜鸟教程 w3cschool.cc false
```

多变量声明

```
//类型相同多个变量, 非全局变量
var vname1, vname2, vname3 type
vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 //和python很像,不需要显示声明类型，自动推断

vname1, vname2, vname3 := v1, v2, v3 //出现在:=左侧的变量不应该是已经被声明过的，否则会导致编译错误

// 这种因式分解关键字的写法一般用于声明全局变量
var (
    vname1 v_type1
    vname2 v_type2
)
```

实例如下：

```
package main

var x, y int
var (  // 这种因式分解关键字的写法一般用于声明全局变量
    a int
    b bool
)

var c, d int = 1, 2
var e, f = 123, "hello"

//这种不带声明格式的只能在函数体中出现
//g, h := 123, "hello"

func main(){
    g, h := 123, "hello"
    println(x, y, a, b, c, d, e, f, g, h)
}
```

以上实例执行结果为：

```
0 0 0 false 1 2 123 hello 123 hello
```

值类型和引用类型

所有像 int、float、bool 和 string 这些基本类型都属于值类型，使用这些类型的变量直接指向存在内存中的值：

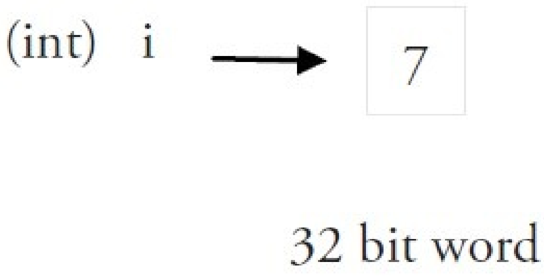


Fig 4.1: Value type

当使用等号 `=` 将一个变量的值赋值给另一个变量时，如：`j = i`，实际上是在内存中将 i 的值进行了拷贝：

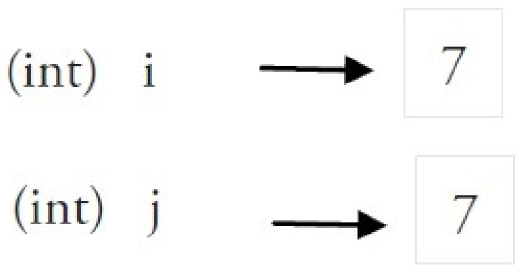


Fig 4.2: Assignment of value types

你可以通过 &i 来获取变量 i 的内存地址，例如：0xf840000040（每次的地址都可能不一样）。值类型的变量的值存储在栈中。

内存地址会根据机器的不同而有所不同，甚至相同的程序在不同的机器上执行后也会有不同的内存地址。因为每台机器可能有不同的存储器布局，并且位置分配也可能不同。

更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。

一个引用类型的变量 r1 存储的是 r1 的值所在的内存地址（数字），或内存地址中第一个字所在的位置。

Fig 4.3: Reference types and assignment

这个内存地址为称之为指针，这个指针实际上也被存在另外的某一个字中。

同一个引用类型的指针指向的多个字可以是在连续的内存地址中（内存布局是连续的），这也是计算效率最高的一种存储形式；也可以将这些字分散存放在内存中，每个字都指示了下一个字所在的内存地址。

当使用赋值语句 r2 = r1 时，只有引用（地址）被复制。

如果 r1 的值被改变了，那么这个值的所有引用都会指向被修改后的内容，在这个例子中，r2 也会受到影响。

简短形式，使用 := 赋值操作符

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，声明语句写上 var 关键字其实是显得有些多余了，因此我们可以将它们简写为 a := 50 或 b := false。

a 和 b 的类型（int 和 bool）将由编译器自动推断。

这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

注意事项

如果在相同的代码块中，我们不可以再次对于相同名称的变量使用初始化声明，例如：a := 20 就是不被允许的，编译器会提示错误 no new variables on left side of :=，但是 a = 20 是可以的，因为这是给相同的变量赋予一个新的值。

如果你在定义变量 a 之前使用它，则会得到编译错误 undefined: a。

如果你声明了一个局部变量却没有在相同的代码块中使用它，同样会得到编译错误，例如下面这个例子当中的变量 a：

```
func main() {
   var a string = "abc"
   fmt.Println("hello, world")
}
```

尝试编译这段代码将得到错误 **a declared and not used**。

此外，单纯地给 a 赋值也是不够的，这个值必须被使用，所以使用

```
fmt.Println("hello, world", a)
```

会移除错误。

但是全局变量是允许声明但不使用。

>

同一类型的多个变量可以声明在同一行，如：

```
var a, b, c int
```

多变量可以在同一行进行赋值，如：

```
a, b, c = 5, 7, "abc"
```

上面这行假设了变量 a，b 和 c 都已经被声明，否则的话应该这样使用：

```
a, b, c := 5, 7, "abc"
```

右边的这些值以相同的顺序赋值给左边的变量，所以 a 的值是 5， b 的值是 7，c 的值是 "abc"。

这被称为 并行 或 同时 赋值。

如果你想要交换两个变量的值，则可以简单地使用 a, b = b, a。

空白标识符 _ 也被用于抛弃值，如值 5 在：_, b = 5, 7 中被抛弃。

_ 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 val 和错误 err 是通过调用 Func1 函数同时得到：val, err = Func1(var1)。

Go 语言常量

常量是一个简单值的标识符，在程序运行时，不会被修改的量。

常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：

```
const identifier [type] = value
```

你可以省略类型说明符 [type]，因为编译器可以根据变量的值来推断其类型。

- 显式类型定义： `const b string = "abc"`
- 隐式类型定义： `const b = "abc"`

多个相同类型的声明可以简写为：

```
const c_name1, c_name2 = value1, value2
```

以下实例演示了常量的应用：

```
package main

import "fmt"

func main() {
   const LENGTH int = 10
   const WIDTH int = 5
   var area int
   const a, b, c = 1, false, "str" //多重赋值

   area = LENGTH * WIDTH
   fmt.Printf("面积为 : %d", area)
   println()
   println(a, b, c)
}
```

以上实例运行结果为：

```
面积为 : 50
1 false str
```

常量还可以用作枚举：

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

数字 0、1 和 2 分别代表未知性别、女性和男性。

常量可以用len(), cap(), unsafe.Sizeof()常量计算表达式的值。常量表达式中，函数必须是内置函数，否则编译不过：

```
package main

import "unsafe"
const (
    a = "abc"
    b = len(a)
    c = unsafe.Sizeof(a)
)

func main(){
    println(a, b, c)
}
```

以上实例运行结果为：

```
abc 3 16
```

iota

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

在每一个const关键字出现时，被重置为0，然后再下一个const出现之前，每出现一次iota，其所代表的数字会自动增加1。

iota 可以被用作枚举值：

```
const (
    a = iota
    b = iota
    c = iota
)
```

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2 可以简写为如下形式：

```
const (
    a = iota
    b
    c
)
```

iota 用法

```
package main

import "fmt"

func main() {
    const (
            a = iota   //0
            b          //1
            c          //2
            d = "ha"   //独立值，iota += 1
            e          //"ha"   iota += 1
            f = 100    //iota +=1
            g          //100  iota +=1
            h = iota   //7,恢复计数
            i          //8
    )
    fmt.Println(a,b,c,d,e,f,g,h,i)
}
```

以上实例运行结果为：

```
0 1 2 ha ha 100 100 7 8
```

再看个有趣的的 iota 实例：

```
package main

import "fmt"
const (
    i=1<<iota
    j=3<<iota
    k
    l
)

func main() {
    fmt.Println("i=",i)
    fmt.Println("j=",j)
    fmt.Println("k=",k)
    fmt.Println("l=",l)
}
```

以上实例运行结果为：

```
i= 1
j= 6
k= 12
l= 24
```

iota表示从0开始自动加1，所以i=1<<0,j=3<<1（<<表示左移的意思），即：i=1,j=6，这没问题，关键在k和l，从输出结果看，k=3<<2，l=3<<3。

Go 语言运算符

运算符用于在程序运行时执行数学或逻辑运算。

Go 语言内置的运算符有：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 其他运算符

接下来让我们来详细看看各个运算符的介绍。

算术运算符

下表列出了所有Go语言的算术运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
+	相加	A + B 输出结果 30
-	相减	A - B 输出结果 -10
*	相乘	A * B 输出结果 200
/	相除	B / A 输出结果 2
%	求余	B % A 输出结果 0
++	自增	A++ 输出结果 11
--	自减	A-- 输出结果 9

以下实例演示了各个算术运算符的用法：

```
package main

import "fmt"

func main() {

   var a int = 21
   var b int = 10
   var c int

   c = a + b
   fmt.Printf("第一行 - c 的值为 %d\n", c )
   c = a - b
   fmt.Printf("第二行 - c 的值为 %d\n", c )
   c = a * b
   fmt.Printf("第三行 - c 的值为 %d\n", c )
   c = a / b
   fmt.Printf("第四行 - c 的值为 %d\n", c )
   c = a % b
   fmt.Printf("第五行 - c 的值为 %d\n", c )
   a++
   fmt.Printf("第六行 - c 的值为 %d\n", a )
   a--
   fmt.Printf("第七行 - c 的值为 %d\n", a )
}
```

以上实例运行结果：

```
第一行 - c 的值为 31
第二行 - c 的值为 11
第三行 - c 的值为 210
第四行 - c 的值为 2
第五行 - c 的值为 1
第六行 - c 的值为 22
第七行 - c 的值为 21
```

关系运算符

下表列出了所有Go语言的关系运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
==	检查两个值是否相等，如果相等返回 True 否则返回 False。	(A == B) 为 False
!=	检查两个值是否不相等，如果不相等返回 True 否则返回 False。	(A != B) 为 True
>	检查左边值是否大于右边值，如果是返回 True 否则返回 False。	(A > B) 为 False
<	检查左边值是否小于右边值，如果是返回 True 否则返回 False。	(A < B) 为 True
>=	检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。	(A >= B) 为 False
<=	检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。	(A <= B) 为 True

以下实例演示了关系运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 21
   var b int = 10

   if( a == b ) {
      fmt.Printf("第一行 - a 等于 b\n" )
   } else {
      fmt.Printf("第一行 - a 不等于 b\n" )
   }
   if ( a < b ) {
      fmt.Printf("第二行 - a 小于 b\n" )
   } else {
      fmt.Printf("第二行 - a 不小于 b\n" )
   }

   if ( a > b ) {
      fmt.Printf("第三行 - a 大于 b\n" )
   } else {
      fmt.Printf("第三行 - a 不大于 b\n" )
   }
   /* Lets change value of a and b */
   a = 5
   b = 20
   if ( a <= b ) {
      fmt.Printf("第四行 - a 小于等于  b\n" )
   }
   if ( b >= a ) {
      fmt.Printf("第五行 - b 大于等于 b\n" )
   }
}
```

以上实例运行结果：

```
第一行 - a 不等于 b
第二行 - a 不小于 b
第三行 - a 大于 b
第四行 - a 小于等于  b
第五行 - b 大于等于 b
```

逻辑运算符

下表列出了所有Go语言的逻辑运算符。假定 A 值为 True，B 值为 False。

运算符	描述	实例
&&	逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。	(A && B) 为 False
||	逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。	(A || B) 为 True
!	逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。	!(A && B) 为 True

以下实例演示了逻辑运算符的用法：

```
package main

import "fmt"

func main() {
   var a bool = true
   var b bool = false
   if ( a && b ) {
      fmt.Printf("第一行 - 条件为 true\n" )
   }
   if ( a || b ) {
      fmt.Printf("第二行 - 条件为 true\n" )
   }
   /* 修改 a 和 b 的值 */
   a = false
   b = true
   if ( a && b ) {
      fmt.Printf("第三行 - 条件为 true\n" )
   } else {
      fmt.Printf("第三行 - 条件为 false\n" )
   }
   if ( !(a && b) ) {
      fmt.Printf("第四行 - 条件为 true\n" )
   }
}
```

以上实例运行结果：

```
第二行 - 条件为 true
第三行 - 条件为 false
第四行 - 条件为 true
```

位运算符

位运算符对整数在内存中的二进制位进行操作。

下表列出了位运算符 &, |, 和 ^ 的计算：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假定 A = 60; B = 13; 其二进制数转换为：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

C 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

运算符	描述	实例
&	按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。	(A & B) 结果为 12, 二进制为 0000 1100
|	按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或	(A | B) 结果为 61, 二进制为 0011 1101
^	按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。	(A ^ B) 结果为 49, 二进制为 0011 0001
<<	左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	A << 2 结果为 240 ，二进制为 1111 0000
>>	右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。	A >> 2 结果为 15 ，二进制为 0000 1111

以下实例演示了逻辑运算符的用法：

```
package main

import "fmt"

func main() {

   var a uint = 60    /* 60 = 0011 1100 */
   var b uint = 13    /* 13 = 0000 1101 */
   var c uint = 0

   c = a & b       /* 12 = 0000 1100 */
   fmt.Printf("第一行 - c 的值为 %d\n", c )

   c = a | b       /* 61 = 0011 1101 */
   fmt.Printf("第二行 - c 的值为 %d\n", c )

   c = a ^ b       /* 49 = 0011 0001 */
   fmt.Printf("第三行 - c 的值为 %d\n", c )

   c = a << 2     /* 240 = 1111 0000 */
   fmt.Printf("第四行 - c 的值为 %d\n", c )

   c = a >> 2     /* 15 = 0000 1111 */
   fmt.Printf("第五行 - c 的值为 %d\n", c )
}
```

以上实例运行结果：

```
第一行 - c 的值为 12
第二行 - c 的值为 61
第三行 - c 的值为 49
第四行 - c 的值为 240
第五行 - c 的值为 15
```

赋值运算符

下表列出了所有Go语言的赋值运算符。

运算符	描述	实例
=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C = *A* 等于 *C* = *C* A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移赋值	C <<= 2 等于 C = C << 2
>>=	右移赋值	C >>= 2 等于 C = C >> 2
&=	位逻辑与赋值	C &= 2 等于 C = C & 2
^=	位逻辑或赋值	C ^= 2 等于 C = C ^ 2
|=	位逻辑异或赋值	C |= 2 等于 C = C | 2

以下实例演示了赋值运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 21
   var c int

   c =  a
   fmt.Printf("第 1 行 - =  运算符实例，c 值为 = %d\n", c )

   c +=  a
   fmt.Printf("第 2 行 - += 运算符实例，c 值为 = %d\n", c )

   c -=  a
   fmt.Printf("第 3 行 - -= 运算符实例，c 值为 = %d\n", c )

   c *=  a
   fmt.Printf("第 4 行 - *= 运算符实例，c 值为 = %d\n", c )

   c /=  a
   fmt.Printf("第 5 行 - /= 运算符实例，c 值为 = %d\n", c )

   c  = 200;

   c <<=  2
   fmt.Printf("第 6行  - <<= 运算符实例，c 值为 = %d\n", c )

   c >>=  2
   fmt.Printf("第 7 行 - >>= 运算符实例，c 值为 = %d\n", c )

   c &=  2
   fmt.Printf("第 8 行 - &= 运算符实例，c 值为 = %d\n", c )

   c ^=  2
   fmt.Printf("第 9 行 - ^= 运算符实例，c 值为 = %d\n", c )

   c |=  2
   fmt.Printf("第 10 行 - |= 运算符实例，c 值为 = %d\n", c )

}
```

以上实例运行结果：

```
第 1 行 - =  运算符实例，c 值为 = 21
第 2 行 - += 运算符实例，c 值为 = 42
第 3 行 - -= 运算符实例，c 值为 = 21
第 4 行 - *= 运算符实例，c 值为 = 441
第 5 行 - /= 运算符实例，c 值为 = 21
第 6行  - <<= 运算符实例，c 值为 = 800
第 7 行 - >>= 运算符实例，c 值为 = 200
第 8 行 - &= 运算符实例，c 值为 = 0
第 9 行 - ^= 运算符实例，c 值为 = 2
第 10 行 - |= 运算符实例，c 值为 = 2
```

其他运算符

下表列出了Go语言的其他运算符。

运算符	描述	实例
&	返回变量存储地址	&a; 将给出变量的实际地址。
*	指针变量。	*a; 是一个指针变量

以下实例演示了其他运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 4
   var b int32
   var c float32
   var ptr *int

   /* 运算符实例 */
   fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );
   fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );
   fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

   /*  & 和 * 运算符实例 */
   ptr = &a    /* 'ptr' 包含了 'a' 变量的地址 */
   fmt.Printf("a 的值为  %d\n", a);
   fmt.Printf("*ptr 为 %d\n", *ptr);
}
```

以上实例运行结果：

```
第 1 行 - a 变量类型为 = int
第 2 行 - b 变量类型为 = int32
第 3 行 - c 变量类型为 = float32
a 的值为  4
*ptr 为 4
```

运算符优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

优先级	运算符
7	^ !
6	* / % << >> & &^
5	+ - | ^
4	== != < <= >= >
3	<-
2	&&
1	||

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

以上实例运行结果：

```
package main

import "fmt"

func main() {
   var a int = 20
   var b int = 10
   var c int = 15
   var d int = 5
   var e int;

   e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   fmt.Printf("(a + b) * c / d 的值为 : %d\n",  e );

   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   fmt.Printf("((a + b) * c) / d 的值为  : %d\n" ,  e );

   e = (a + b) * (c / d);   // (30) * (15/5)
   fmt.Printf("(a + b) * (c / d) 的值为  : %d\n",  e );

   e = a + (b * c) / d;     //  20 + (150/5)
   fmt.Printf("a + (b * c) / d 的值为  : %d\n" ,  e );
}
```

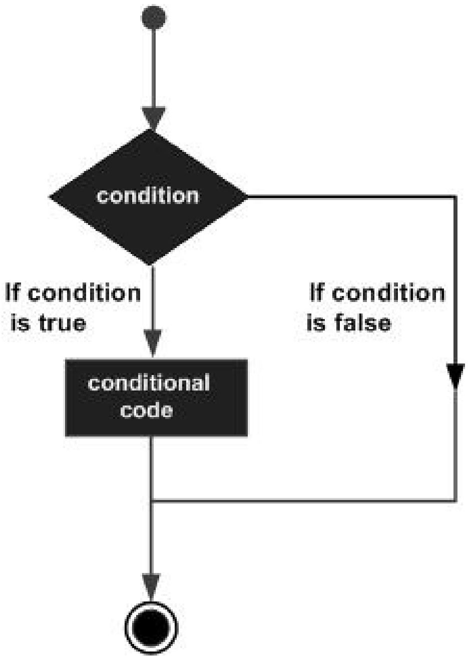
以上实例运行结果：

```
(a + b) * c / d 的值为 : 90
((a + b) * c) / d 的值为  : 90
(a + b) * (c / d) 的值为  : 90
a + (b * c) / d 的值为  : 50
```

Go 语言条件语句

条件语句需要开发者通过指定一个或多个条件，并通过测试条件是否为 true 来决定是否执行指定语句，并在条件为 false 的情况在执行另外的语句。

下图展示了程序语言中条件语句的结构：



Go 语言提供了以下几种条件判断语句：

语句	描述
if 语句	**if 语句** 由一个布尔表达式后紧跟一个或多个语句组成。
if...else 语句	**if 语句** 后可以使用可选的 **else 语句**, else 语句中的表达式在布尔表达式为 false 时执行。
if 嵌套语句	你可以在 **if** 或 **else if** 语句中嵌入一个或多个 **if** 或 **else if** 语句。
switch 语句	**switch** 语句用于基于不同条件执行不同动作。
select 语句	**select** 语句类似于 **switch** 语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。

Go 语言 if 语句

if 语句由布尔表达式后紧跟一个或多个语句组成。

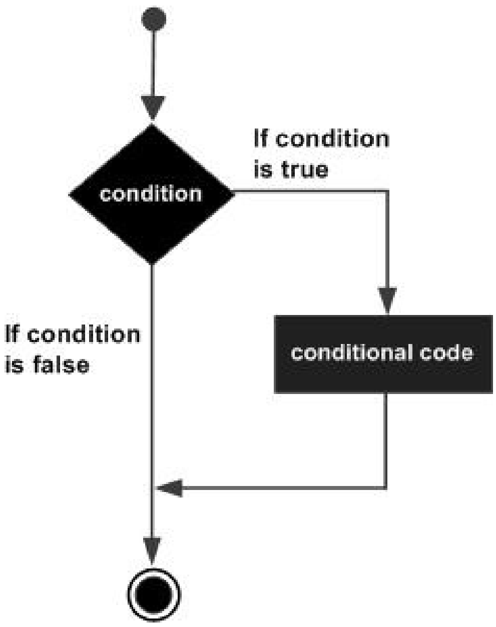
语法

Go 编程语言中 if 语句的语法如下：

```
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
}
```

If 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则不执行。

流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* 使用 if 语句判断布尔表达式 */
   if a < 20 {
       /* 如果条件为 true 则执行以下语句 */
       fmt.Printf("a 小于 20\n" )
   }
   fmt.Printf("a 的值为 : %d\n", a)
}
```

以上代码执行结果为：

```
a 小于  20
a 的值为  :  10
```

Go 语言 if...else 语句

if 语句 后可以使用可选的 else 语句, else 语句中的表达式在布尔表达式为 false 时执行。

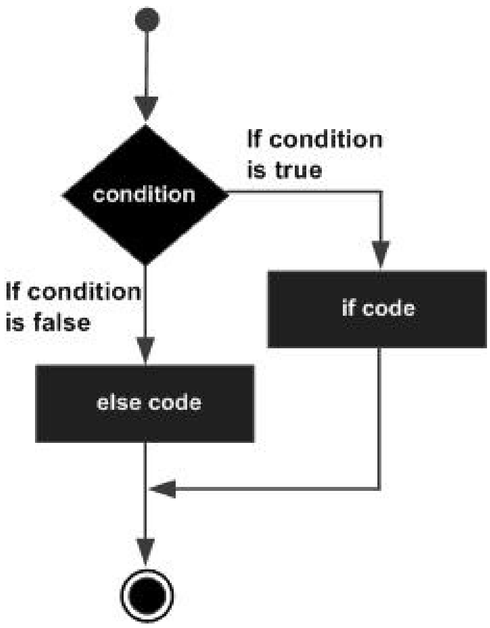
语法

Go 编程语言中 if...else 语句的语法如下：

```
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
} else {
  /* 在布尔表达式为 false 时执行 */
}
```

If 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则执行 else 语句块。

流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 局部变量定义 */
   var a int = 100;

   /* 判断布尔表达式 */
   if a < 20 {
       /* 如果条件为 true 则执行以下语句 */
       fmt.Printf("a 小于 20\n" );
   } else {
       /* 如果条件为 false 则执行以下语句 */
       fmt.Printf("a 不小于 20\n" );
   }
   fmt.Printf("a 的值为 : %d\n", a);

}
```

以上代码执行结果为：

```
a 不小于  20
a 的值为  :  100
```

Go 语言 if 语句嵌套

你可以在 if 或 else if 语句中嵌入一个或多个 if 或 else if 语句。

语法

Go 编程语言中 if...else 语句的语法如下：

```
if 布尔表达式 1 {
   /* 在布尔表达式 1 为 true 时执行 */
   if 布尔表达式 2 {
      /* 在布尔表达式 2 为 true 时执行 */
   }
}
```

你可以以同样的方式在 if 语句中嵌套 **else if...else** 语句

实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200

   /* 判断条件 */
   if a == 100 {
       /* if 条件语句为 true 执行 */
       if b == 200 {
          /* if 条件语句为 true 执行 */
          fmt.Printf("a 的值为 100 ， b 的值为 200\n" );
       }
   }
   fmt.Printf("a 值为 : %d\n", a );
   fmt.Printf("b 值为 : %d\n", b );
}
```

以上代码执行结果为：

```
a 的值为  100 ， b 的值为  200
a 值为 :  100
b 值为 :  200
```

Go 语言 switch 语句

switch 语句用于基于不同条件执行不同动作，每一个 case 分支都是唯一的，从上直下逐一测试，直到匹配为止。。

switch 语句执行的过程从上至下，直到找到匹配项，匹配项后面也不需要再加break

语法

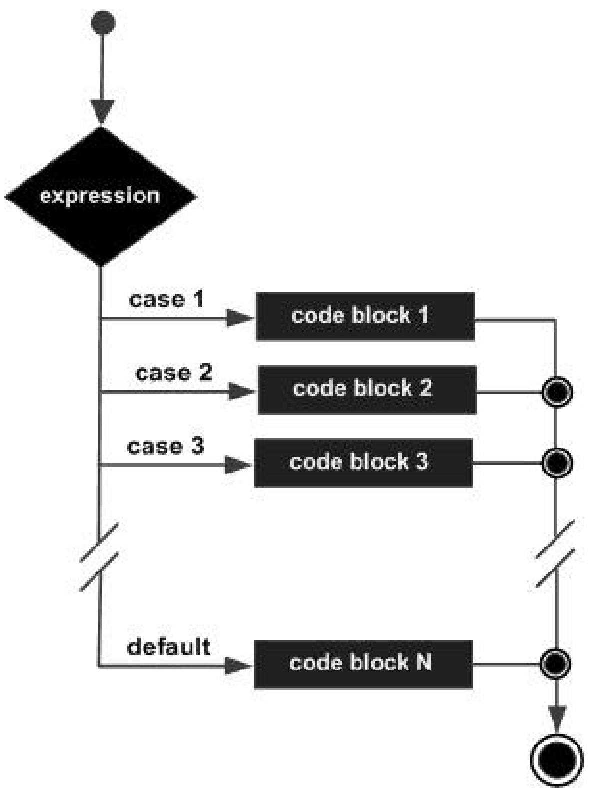
Go 编程语言中 switch 语句的语法如下：

```
switch var1 {
    case val1:
        ...
    case val2:
        ...
    default:
        ...
}
```

变量 var1 可以是任何类型，而 val1 和 val2 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同的类型；或者最终结果为相同类型的表达式。

您可以同时测试多个可能符合条件的值，使用逗号分割它们，例如：case val1, val2, val3。

流程图：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var grade string = "B"
   var marks int = 90

   switch marks {
      case 90: grade = "A"
      case 80: grade = "B"
      case 50,60,70 : grade = "C"
      default: grade = "D"
   }

   switch {
      case grade == "A" :
         fmt.Printf("优秀!\n" )
      case grade == "B", grade == "C" :
         fmt.Printf("良好\n" )
      case grade == "D" :
         fmt.Printf("及格\n" )
      case grade == "F":
         fmt.Printf("不及格\n" )
      default:
         fmt.Printf("差\n" );
   }
   fmt.Printf("你的等级是 %s\n", grade );
}
```

以上代码执行结果为：

```
优秀!
你的等级是 A
```

Type Switch

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

Type Switch 语法格式如下：

```
switch x.(type){
    case type:
       statement(s);
    case type:
       statement(s);
    /* 你可以定义任意个数的case */
    default: /* 可选 */
       statement(s);
}
```

实例

```
package main

import "fmt"

func main() {
   var x interface{}

   switch i := x.(type) {
      case nil:
         fmt.Printf(" x 的类型 :%T",i)
      case int:
         fmt.Printf("x 是 int 型")
      case float64:
         fmt.Printf("x 是 float64 型")
      case func(int) float64:
         fmt.Printf("x 是 func(int) 型")
      case bool, string:
         fmt.Printf("x 是 bool 或 string 型" )
      default:
         fmt.Printf("未知型")
   }
}
```

以上代码执行结果为：

```
x 的类型  :<nil>
```

Go 语言 select 语句

select是Go中的一个控制结构，类似于用于通信的switch语句。每个case必须是一个通信操作，要么是发送要么是接收。

select随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。一个默认的子句应该总是可运行的。

语法

Go 编程语言中 select 语句的语法如下：

```
select {
    case communication clause  :
       statement(s);
    case communication clause  :
       statement(s);
    /* 你可以定义任意数量的 case */
    default : /* 可选 */
       statement(s);
}
```

以下描述了 select 语句的语法：

- 每个case都必须是一个通信
- 所有channel表达式都会被求值
- 所有被发送的表达式都会被求值
- 如果任意某个通信可以进行，它就执行；其他被忽略。
- 如果有多个case都可以运行，Select会随机公平地选出一个执行。其他不会执行。 否则：
    1. 如果有default子句，则执行该语句。
    2. 如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。

实例

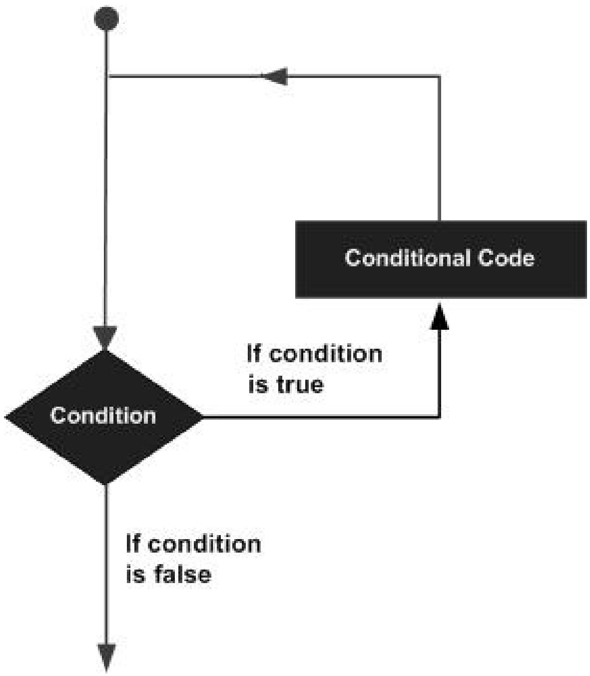
```
package main

import "fmt"

func main() {
   var c1, c2, c3 chan int
   var i1, i2 int
   select {
      case i1 = <-c1:
         fmt.Printf("received ", i1, " from c1\n")
      case c2 <- i2:
         fmt.Printf("sent ", i2, " to c2\n")
      case i3, ok := (<-c3):  // same as: i3, ok := <-c3
         if ok {
            fmt.Printf("received ", i3, " from c3\n")
         } else {
            fmt.Printf("c3 is closed\n")
         }
      default:
         fmt.Printf("no communication\n")
   }
}
```

以上代码执行结果为：

```
no communication
```

Go 语言循环语句

在不少实际问题中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。



以下为大多编程语言循环程序的流程图：

Go 语言提供了以下几种类型循环处理语句：

循环类型	描述
for 循环	重复执行语句块
循环嵌套	在 for 循环中嵌套一个或多个 for 循环

循环控制语句

循环控制语句可以控制循环体内语句的执行过程。

GO 语言支持以下几种循环控制语句：

控制语句	描述
break 语句	经常用于中断当前 for 循环或跳出 switch 语句
continue 语句	跳过当前循环的剩余语句，然后继续进行下一轮循环。
goto 语句	将控制转移到被标记的语句。

无限循环

如过循环中条件语句永远不为 false 则会进行无限循环，我们可以通过 for 循环语句中只设置一个条件表达式来执行无限循环：

```
package main

import "fmt"

func main() {
    for true  {
        fmt.Printf("这是无限循环。\n");
    }
}
```

Go 语言 for 循环

for循环是一个循环控制结构，可以执行指定次数的循环。

语法

Go语言的For循环油3中形式，只有其中的一种使用分号。

和 C 语言的 for 一样：

```
for init; condition; post {  }
```

和 C 的 while 一样：

```
for condition {  }
```

和 C 的 for(;;) 一样：

```
for  {  }
```

- init： 一般为赋值表达式，给控制变量赋初值；
- condition： 关系表达式或逻辑表达式，循环控制条件；
- post： 一般为赋值表达式，给控制变量增量或减量。

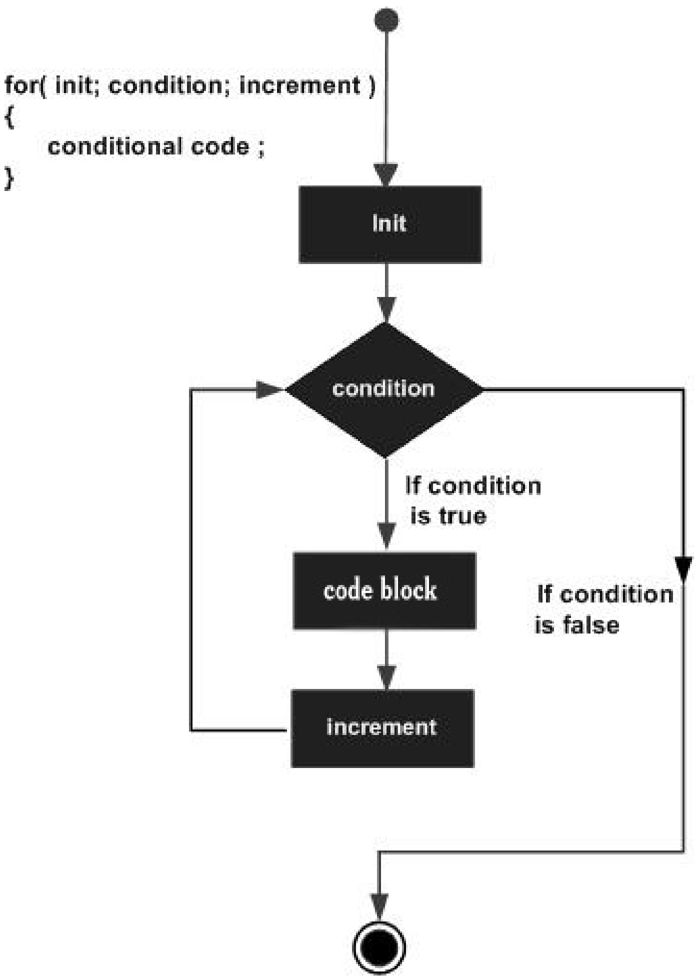
for语句执行过程如下：

- ①先对表达式1赋初值；
- ②判别赋值表达式 init 是否满足给定条件，若其值为真，满足循环条件，则执行循环体内语句，然后执行 post，进入第二次循环，再判别 condition；否则判断 condition 的值为假，不满足条件，就终止for循环，执行循环体外语句。

for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

```
for key, value := range oldMap {
    newMap[key] = value
}
```

for语句语法流程如下图所示：



实例

```
package main

import "fmt"

func main() {

    var b int = 15
    var a int

    numbers := [6]int{1, 2, 3, 5}

    /* for 循环 */
    for a := 0; a < 10; a++ {
       fmt.Printf("a 的值为: %d\n", a)
    }

    for a < b {
       a++
       fmt.Printf("a 的值为: %d\n", a)
       }

    for i,x:= range numbers {
       fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)
    }
}
```

以上实例运行输出结果为:

```
a 的值为: 0
a 的值为: 1
a 的值为: 2
a 的值为: 3
a 的值为: 4
a 的值为: 5
a 的值为: 6
a 的值为: 7
a 的值为: 8
a 的值为: 9
a 的值为: 1
a 的值为: 2
a 的值为: 3
a 的值为: 4
a 的值为: 5
a 的值为: 6
a 的值为: 7
a 的值为: 8
a 的值为: 9
a 的值为: 10
a 的值为: 11
a 的值为: 12
a 的值为: 13
a 的值为: 14
a 的值为: 15
第 0 位 x 的值 = 1
第 1 位 x 的值 = 2
第 2 位 x 的值 = 3
第 3 位 x 的值 = 5
第 4 位 x 的值 = 0
第 5 位 x 的值 = 0
```

Go 语言循环嵌套

Go 语言允许用户在循环内使用循环。接下来我们将为大家介绍嵌套循环的使用。

语法

以下为 Go 语言嵌套循环的格式：

```
for [condition |  ( init; condition; increment ) | Range]
{
   for [condition |  ( init; condition; increment ) | Range]
   {
      statement(s);
   }
   statement(s);
}
```

实例

以下实例使用循环嵌套来输出 2 到 100 间的素数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var i, j int

   for i=2; i < 100; i++ {
      for j=2; j <= (i/j); j++ {
         if(i%j==0) {
            break; // 如果发现因子，则不是素数
         }
      }
      if(j > (i/j)) {
         fmt.Printf("%d  是素数\n", i);
      }
   }
}
```

以上实例运行输出结果为:

```
2  是素数
3  是素数
5  是素数
7  是素数
11  是素数
13  是素数
17  是素数
19  是素数
23  是素数
29  是素数
31  是素数
37  是素数
41  是素数
43  是素数
47  是素数
53  是素数
59  是素数
61  是素数
67  是素数
71  是素数
73  是素数
79  是素数
83  是素数
89  是素数
97  是素数
```

Go 语言 break 语句

Go语言循环语句

Go 语言中 break 语句用于以下两方面：

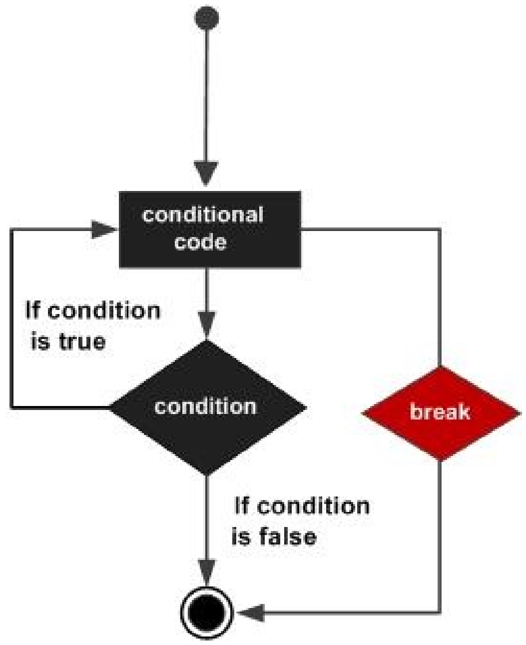
1. 用于循环语句中跳出循环，并开始执行循环之后的语句。
2. break在switch（开关语句）中在执行一条case后跳出语句的作用。

语法

break 语法格式如下：

```
break;
```

break 语句流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* for 循环 */
   for a < 20 {
      fmt.Printf("a 的值为 : %d\n", a);
      a++;
      if a > 15 {
         /* 使用 break 语句跳出循环 */
         break;
      }
   }
}
```

以上实例执行结果为：

```
a 的值为 : 10
a 的值为 : 11
a 的值为 : 12
a 的值为 : 13
a 的值为 : 14
a 的值为 : 15
```

Go语言循环语句

Go 语言 continue 语句

Go 语言的 continue 语句 有点像 break 语句。但是 continue 不是跳出循环，而是跳过当前循环执行下一次循环语句。

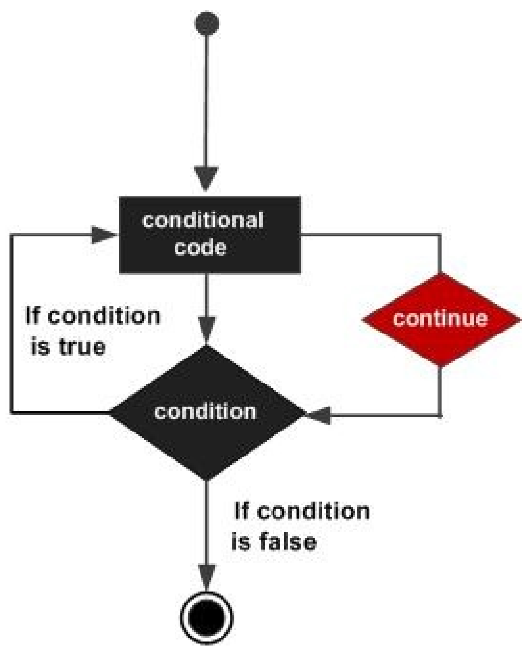
for 循环中，执行 continue 语句会触发for增量语句的执行。

语法

continue 语法格式如下：

```
continue;
```

break 语句流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* for 循环 */
   for a < 20 {
      if a == 15 {
         /* 跳过此次循环 */
         a = a + 1;
         continue;
      }
      fmt.Printf("a 的值为 : %d\n", a);
      a++;
   }
}
```

以上实例执行结果为：

```
a 的值为 : 10
a 的值为 : 10
a 的值为 : 11
a 的值为 : 12
a 的值为 : 13
a 的值为 : 14
a 的值为 : 16
a 的值为 : 17
a 的值为 : 18
a 的值为 : 19
```

Go 语言 goto 语句

Go 语言的 goto 语句可以无条件地转移到过程中指定的行。

goto语句通常与条件语句配合使用。可用来实现条件转移， 构成循环，跳出循环体等功能。

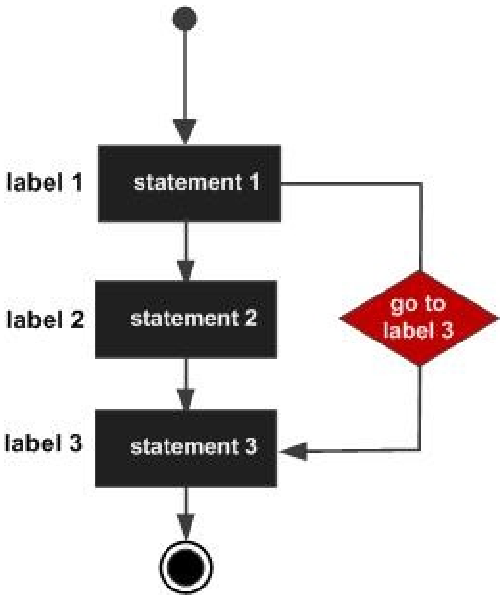
但是，在结构化程序设计中一般不主张使用goto语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。

语法

goto 语法格式如下：

```
goto label;
..
.
label: statement;
```

break 语句流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* 循环 */
   LOOP: for a < 20 {
      if a == 15 {
         /* 跳过迭代 */
         a = a + 1
         goto LOOP
      }
      fmt.Printf("a的值为 : %d\n", a)
      a++
   }
}
```

以上实例执行结果为：

```
a的值为 : 10
a的值为 : 11
a的值为 : 12
a的值为 : 13
a的值为 : 14
a的值为 : 16
a的值为 : 17
a的值为 : 18
a的值为 : 19
```

Go 语言函数

函数是基本的代码块，用于执行一个任务。

Go 语言最少有个 main() 函数。

你可以通过函数来划分不同功能，逻辑上每个函数执行的是指定的任务。

函数声明告诉了编译器函数的名称，返回类型，和参数。

Go 语言标准库提供了多种可动用的内置的函数。例如，len() 函数可以接受不同类型参数并返回该类型的长度。如果我们传入的是字符串则返回字符串的长度，如果传入的是数字，则返回数组中包含的函数个数。

函数定义

Go 语言函数定义格式如下：

```
func function_name( [parameter list] ) [return_types] {
   函数体
}
```

函数定义解析：

- func：函数由 func 开始声明
- function_name：函数名称，函数名和参数列表一起构成了函数签名。
- parameter list]：参数列表，参数就像一个占位符，当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。参数列表指定的是参数类型、顺序、及参数个数。参数是可选的，也就是说函数也可以不包含参数。
- return_types：返回类型，函数返回一列值。return_types 是该列值的数据类型。有些功能不需要返回值，这种情况下 return_types 不是必须的。
- 函数体：函数定义的代码集合。

实例

以下实例为 max() 函数的代码，该函数传入两个整型参数 num1 和 num2，并返回这两个参数的最大值：

```
/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 声明局部变量 */
   result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

函数调用

当创建函数时，你定义了函数需要做什么，通过调用改函数来执行指定任务。

调用函数，向函数传递参数，并返回值，例如：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200
   var ret int

   /* 调用函数并返回最大值 */
   ret = max(a, b)

   fmt.Printf( "最大值是 : %d\n", ret )
}

/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 定义局部变量 */
   var result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

以上实例在 main() 函数中调用 max（）函数，执行结果为：

```
最大值是 :  200
```

函数返回多个值

Go 函数可以返回多个值，例如：

```
package main

import "fmt"

func swap(x, y string) (string, string) {
   return y, x
}

func main() {
   a, b := swap("Mahesh", "Kumar")
   fmt.Println(a, b)
}
```

以上实例执行结果为：

```
Kumar  Mahesh
```

函数参数

函数如果使用参数，该变量可称为函数的形参。

形参就像定义在函数体内的局部变量。

调用函数，可以通过两种方式来传递参数：

传递类型	描述
值传递	值传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。
引用传递	引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

函数用法

函数用法	描述
函数作为值	函数定义后可作为值来使用
闭包	闭包是匿名函数，可在动态编程中使用
方法	方法就是一个包含了接受者的函数

Go 语言函数值传递值

传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

以下定义了 swap() 函数：

```
/* 定义相互交换值的函数 */
func swap(x, y int) int {
   var temp int

   temp = x /* 保存 x 的值 */
   x = y    /* 将 y 值赋给 x */
   y = temp /* 将 temp 值赋给 y*/

   return temp;
}
```

接下来，让我们使用值传递来调用 swap() 函数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200

   fmt.Printf("交换前 a 的值为 : %d\n", a )
   fmt.Printf("交换前 b 的值为 : %d\n", b )

   /* 通过调用函数来交换值 */
   swap(a, b)

   fmt.Printf("交换后 a 的值 : %d\n", a )
   fmt.Printf("交换后 b 的值 : %d\n", b )
}

/* 定义相互交换值的函数 */
func swap(x, y int) int {
   var temp int

   temp = x /* 保存 x 的值 */
   x = y    /* 将 y 值赋给 x */
   y = temp /* 将 temp 值赋给 y*/

   return temp;
}
```

以下代码执行结果为：

```
交换前 a 的值为 : 100
交换前 b 的值为 : 200
交换后 a 的值 : 100
交换后 b 的值 : 200
```

Go 语言函数引用传递值

引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

引用传递指针参数传递到函数内，以下是交换函数 swap() 使用了引用传递：

```
/* 定义交换值函数*/
func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保持 x 地址上的值 */
   *x = *y      /* 将 y 值赋给 x */
   *y = temp    /* 将 temp 值赋给 y */
}
```

以下我们通过使用引用传递来调用 swap() 函数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int= 200

   fmt.Printf("交换前，a 的值 : %d\n", a )
   fmt.Printf("交换前，b 的值 : %d\n", b )

   /* 调用 swap() 函数
   * &a 指向 a 指针，a 变量的地址
   * &b 指向 b 指针，b 变量的地址
   */
   swap(&a, &b)

   fmt.Printf("交换后，a 的值 : %d\n", a )
   fmt.Printf("交换后，b 的值 : %d\n", b )
}

func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保存 x 地址上的值 */
   *x = *y      /* 将 y 值赋给 x */
   *y = temp    /* 将 temp 值赋给 y */
}
```

以上代码执行结果为：

```
交换前，a 的值 : 100
交换前，b 的值 : 200
交换后，a 的值 : 200
交换后，b 的值 : 100
```

Go 语言函数作为值

Go 语言可以很灵活的创建函数，并作为值使用。以下实例中我们在定义的函数中初始化一个变量，该函数仅仅是为了使用内置函数 math.sqrt()，实例为：

```
package main

import (
   "fmt"
   "math"
)

func main(){
   /* 声明函数变量 */
   getSquareRoot := func(x float64) float64 {
      return math.Sqrt(x)
   }

   /* 使用函数 */
   fmt.Println(getSquareRoot(9))

}
```

以上代码执行结果为：

```
3
```

Go 语言函数闭包

Go 语言支持匿名函数，可作为闭包。匿名函数是一个"内联"语句或表达式。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

以下实例中，我们创建了函数 getSequence() ，返回另外一个函数。该函数的目的是在闭包中递增 i 变量，代码如下：

```
package main

import "fmt"

func getSequence() func() int {
   i:=0
   return func() int {
      i+=1
     return i
   }
}

func main(){
   /* nextNumber 为一个函数，函数 i 为 0 */
   nextNumber := getSequence()

   /* 调用 nextNumber 函数，i 变量自增 1 并返回 */
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())

   /* 创建新的函数 nextNumber1，并查看结果 */
   nextNumber1 := getSequence()
   fmt.Println(nextNumber1())
   fmt.Println(nextNumber1())
}
```

以上代码执行结果为：

```
1
2
3
1
2
```

Go 语言函数方法

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。语法格式如下：

```
func (variable_name variable_data_type) function_name() [return_type]{
   /* 函数体*/
}
```

下面定义一个结构体类型和该类型的一个方法：

```
package main

import (
   "fmt"
)

/* 定义函数 */
type Circle struct {
  radius float64
}

func main() {
  var c1 Circle
  c1.radius = 10.00
  fmt.Println("Area of Circle(c1) = ", c1.getArea())
}

//该 method 属于 Circle 类型对象中的方法
func (c Circle) getArea() float64 {
  //c.radius 即为 Circle 类型对象中的属性
  return 3.14 * c.radius * c.radius
}
```

以上代码执行结果为：

```
Area of Circle(c1)  =  314
```

Go 语言变量作用域

作用域为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。

Go 语言中变量可以在三个地方声明：

- 函数内定义的变量称为局部变量
- 函数外定义的变量称为全局变量
- 函数定义中的变量称为形式参数

接下来让我们具体了解局部变量、全局变量和形式参数。

局部变量

在函数体内声明的变量称之为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

以下实例中 main() 函数使用了局部变量 a, b, c：

```
package main

import "fmt"

func main() {
   /* 声明局部变量 */
   var a, b, c int

   /* 初始化参数 */
   a = 10
   b = 20
   c = a + b

   fmt.Printf ("结果： a = %d, b = %d and c = %d\n", a, b, c)
}
```

以上实例执行输出结果为：

```
结果： a = 10, b = 20 and c = 30
```

全局变量

在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。

全局变量可以在任何函数中使用，以下实例演示了如何使用全局变量：

```
package main

import "fmt"

/* 声明全局变量 */
var g int

func main() {

   /* 声明局部变量 */
   var a, b int

   /* 初始化参数 */
   a = 10
   b = 20
   g = a + b

   fmt.Printf("结果： a = %d, b = %d and g = %d\n", a, b, g)
}
```

以上实例执行输出结果为：

```
结果： a = 10, b = 20 and g = 30
```

Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑。实例如下：

```
package main

import "fmt"

/* 声明全局变量 */
var g int = 20

func main() {
   /* 声明局部变量 */
   var g int = 10

   fmt.Printf ("结果： g = %d\n",  g)
}
```

以上实例执行输出结果为：

```
结果： g = 10
```

形式参数

形式参数会作为函数的局部变量来使用。实例如下：

```
package main

import "fmt"

/* 声明全局变量 */
var a int = 20;

func main() {
   /* main 函数中声明局部变量 */
   var a int = 10
   var b int = 20
   var c int = 0

   fmt.Printf("main()函数中 a = %d\n",  a);
   c = sum( a, b);
   fmt.Printf("main()函数中 c = %d\n",  c);
}

/* 函数定义-两数相加 */
func sum(a, b int) int {
   fmt.Printf("sum() 函数中 a = %d\n",  a);
   fmt.Printf("sum() 函数中 b = %d\n",  b);

   return a + b;
}
```

以上实例执行输出结果为：

```
main()函数中 a = 10
sum() 函数中 a = 10
sum() 函数中 b = 20
main()函数中 c = 30
```

初始化局部和全局变量

不同类型的局部和全局变量默认值为：

数据类型	初始化默认值
int	0
float32	0
pointer	nil

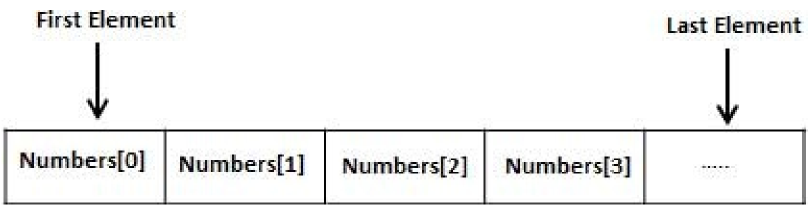
Go 语言数组

Go 语言提供了数组类型的数据结构。

数组是具有相同唯一类型的一组已编号且长度固定的数据项序列，这种类型可以是任意的原始类型例如整形、字符串或者自定义类型。

相对于去声明number0, number1, ..., and number99的变量，使用数组形式numbers[0], numbers[1] ..., numbers[99]更加方便且易于扩展。

数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。



声明数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

```
var variable_name [SIZE] variable_type
```

以上为一维数组的定义方式。数组长度必须是整数且大于 0。例如以下定义了数组 balance 长度为 10 类型为 float32：

```
var balance [10] float32
```

初始化数组

以下演示了数组初始化：

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

初始化数组中 {} 中的元素个数不能大于 [] 中的数字。

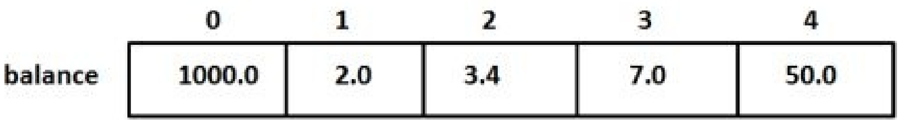
如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小：

```
 var balance = []float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

该实例与上面的实例是一样的，虽然没有设置数组的大小。

```
balance[4] = 50.0
```

以上实例读取了第五个元素。数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。



访问数组元素

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

```
float32 salary = balance[9]
```

以上实例读取了数组balance第10个元素的值。

以下演示了数组完整操作（声明、赋值、访问）的实例：

```
package main

import "fmt"

func main() {
   var n [10]int /* n 是一个长度为 10 的数组 */
   var i,j int

   /* 为数组 n 初始化元素 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* 设置元素为 i + 100 */
   }

   /* 输出每个数组元素的值 */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
   }
}
```

以上实例执行结果如下：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

更多内容

数组对 Go 语言来说是非常重要的，以下我们将介绍数组更多的内容：

内容	描述
多维数组	Go 语言支持多维数组，最简单的多维数组是二维数组
向函数传递数组	你可以像函数传递数组参数

Go 语言多维数组

Go 语言支持多维数组，以下为常用的多维数组声明方式：

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type
```

以下实例声明了三维的整型数组：

```
var threedim [5][10][4]int
```

二维数组

二维数组是最简单的多维数组，二维数组本质上是由一维数组组成的。二维数组定义方式如下：

```
var arrayName [ x ][ y ] variable_type
```

variable_type 为 Go 语言的数据类型，arrayName 为数组名，二维数组可认为是一个表格，x 为行，y 为列，下图演示了一个二维数组 a 为三行四列：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

二维数组中的元素可通过 *a[i][j]* 来访问。

初始化二维数组

多维数组可通过大括号来初始值。以下实例为一个 3 行 4 列的二维数组：

```
a = [3][4]int{
 {0, 1, 2, 3} ,   /*  第一行索引为 0 */
 {4, 5, 6, 7} ,   /*  第二行索引为 1 */
 {8, 9, 10, 11}   /*  第三行索引为 2 */
}
```

访问二维数组

二维数组通过指定坐标来访问。如数组中的行索引与列索引，例如：

```
int val = a[2][3]
```

以上实例访问了二维数组 val 第三行的第四个元素。

二维数组可以使用循环嵌套来输出元素：

```
package main

import "fmt"

func main() {
   /* 数组 - 5 行 2 列*/
   var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}
   var i, j int

   /* 输出数组元素 */
   for  i = 0; i < 5; i++ {
      for j = 0; j < 2; j++ {
         fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )
      }
   }
}
```

以上实例运行输出结果为：

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

Go 语言向函数传递数组

如果你想向函数传递数组参数，你需要在函数定义时，声明形参为数组，我们可以通过以下两种方式来声明：

方式一

形参设定数组大小：

```
void myFunction(param [10]int)
{
.
.
.
}
```

方式二

形参未设定数组大小：

```
void myFunction(param []int)
{
.
.
.
}
```

实例

让我们看下以下实例，实例中函数接收整型数组参数，另一个参数指定了数组元素的个数，并返回平均值：

```
func getAverage(arr []int, int size) float32
{
   var i int
   var avg, sum float32

   for i = 0; i < size; ++i {
      sum += arr[i]
   }

   avg = sum / size

   return avg;
}
```

接下来我们来调用这个函数：

```
package main

import "fmt"

func main() {
   /* 数组长度为 5 */
   var  balance = []int {1000, 2, 3, 17, 50}
   var avg float32

   /* 数组作为参数传递给函数 */
   avg = getAverage( balance, 5 ) ;

   /* 输出返回的平均值 */
   fmt.Printf( "平均值为: %f ", avg );
}
func getAverage(arr []int, size int) float32 {
   var i,sum int
   var avg float32

   for i = 0; i < size;i++ {
      sum += arr[i]
   }

   avg = float32(sum / size)

   return avg;
}
```

以上实例执行输出结果为：

```
平均值为:  214.000000
```

以上实例中我们使用的形参并为设定数组大小。

Go 语言指针

Go 语言中指针是很容易学习的，Go 语言中使用指针可以更简单的执行一些任务。

接下来让我们来一步步学习 Go 语言指针。

我们都知道，变量是一种使用方便的占位符，用于引用计算机内存地址。

Go 语言的取地址符是 &，放到一个变量前使用就会返回相应变量的内存地址。

以下实例演示了变量在内存中地址：

```
package main

import "fmt"

func main() {
   var a int = 10

   fmt.Printf("变量的地址: %x\n", &a  )
}
```

执行以上代码输出结果为：

```
变量的地址: 20818a220
```

现在我们已经了解了什么是内存地址和如何去反问它。接下来我们将具体介绍指针。

什么是指针

一个指针变量可以指向任何一个值的内存地址它指向那个值的内存地址。

类似于变量和常量，在使用指针前你需要声明指针。指针声明格式如下：

```
var var_name *var-type
```

var-type 为指针类型，var_name 为指针变量名，* 号用于指定变量是作为一个指针。以下是有效的指针声明：

```
var ip *int        /* 指向整型*/
var fp *float32    /* 指向浮点型 */
```

本例中这是一个指向 int 和 float32 的指针。

如何使用指针

指针使用流程：

- 定义指针变量。
- 为指针变量赋值。
- 访问指针变量中指向地址的值。

在指针类型前面加上 * 号（前缀）来获取指针所指向的内容。

```
package main

import "fmt"

func main() {
   var a int= 20   /* 声明实际变量 */
   var ip *int        /* 声明指针变量 */

   ip = &a  /* 指针变量的存储地址 */

   fmt.Printf("a 变量的地址是: %x\n", &a  )

   /* 指针变量的存储地址 */
   fmt.Printf("ip 变量的存储地址: %x\n", ip )

   /* 使用指针访问值 */
   fmt.Printf("*ip 变量的值: %d\n", *ip )
}
```

以上实例执行输出结果为：

```
a 变量的地址是: 20818a220
ip 变量的存储地址: 20818a220
*ip 变量的值: 20
```

Go 空指针

当一个指针被定义后没有分配到任何变量时，它的值为 nil。

nil 指针也称为空指针。

nil在概念上和其它语言的null、None、nil、NULL一样，都指代零值或空值。

一个指针变量通常缩写为 ptr。

查看以下实例：

```
package main

import "fmt"

func main() {
   var  ptr *int

   fmt.Printf("ptr 的值为 : %x\n", ptr  )
}
```

以上实例输出结果为：

```
ptr 的值为 : 0
```

空指针判断：

```
if(ptr != nil)     /* ptr 不是空指针 */
if(ptr == nil)    /* ptr 是空指针 */
```

Go指针更多内容

接下来我们将为大家介绍Go语言中更多的指针应用：

内容	描述
Go 指针数组	你可以定义一个指针数组来存储地址
Go 指向指针的指针	Go 支持指向指针的指针
Go 像函数传递指针参数	通过引用或地址传参，在函数调用时可以改变其值

Go 语言指针数组

在我们了解指针数组前，先看个实例，定义了长度为 3 的整型数组：

```
package main

import "fmt"

const MAX int = 3

func main() {

   a := []int{10,100,200}
   var i int

   for i = 0; i < MAX; i++ {
      fmt.Printf("a[%d] = %d\n", i, a[i] )
   }
}
```

以上代码执行输出结果为：

```
a[0] = 10
a[1] = 100
a[2] = 200
```

有一种情况，我们可能需要保存数组，这样我们就需要使用到指针。

以下声明了整型指针数组：

```
var ptr [MAX]*int;
```

ptr 为整型指针数组。因此每个元素都指向了一个值。以下实例的三个整数将存储在指针数组中：

```
package main

import "fmt"

const MAX int = 3

func main() {
   a := []int{10,100,200}
   var i int
   var ptr [MAX]*int;

   for  i = 0; i < MAX; i++ {
      ptr[i] = &a[i] /* 整数地址赋值给指针数组 */
   }

   for  i = 0; i < MAX; i++ {
      fmt.Printf("a[%d] = %d\n", i,*ptr[i] )
   }
}
```

以上代码执行输出结果为：

```
a[0] = 10
a[1] = 100
a[2] = 200
```

Go 语言指向指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。

当定义一个指向指针的指针变量时，第一个指针存放第二个指针的地址，第二个指针存放变量的地址：

指向指针的指针变量声明格式如下：

```
var ptr **int;
```

以上指向指针的指针变量为整型。

访问指向指针的指针变量值需要使用两个 * 号，如下所示：

```
package main

import "fmt"

func main() {

    var a int
    var ptr *int
    var pptr **int

    a = 3000

    /* 指针 ptr 地址 */
    ptr = &a

    /* 指向指针 ptr 地址 */
    pptr = &ptr

    /* 获取 pptr 的值 */
    fmt.Printf("变量 a = %d\n", a )
    fmt.Printf("指针变量 *ptr = %d\n", *ptr )
    fmt.Printf("指向指针的指针变量 **pptr = %d\n", **pptr)
}
```

以上实例执行输出结果为：

```
变量 a = 3000
指针变量 *ptr = 3000
指向指针的指针变量 **pptr = 3000
```

Go 语言指针作为函数参数

Go 语言允许向函数传递指针，志需要在函数定义的参数上设置为指针类型即可。

以下实例演示了如何向函数传递指针，并在函数调用后修改函数内的值，：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int= 200

   fmt.Printf("交换前 a 的值 : %d\n", a )
   fmt.Printf("交换前 b 的值 : %d\n", b )

   /* 调用函数用于交换值
   * &a 指向 a 变量的地址
   * &b 指向 b 变量的地址
   */
   swap(&a, &b);

   fmt.Printf("交换后 a 的值 : %d\n", a )
   fmt.Printf("交换后 b 的值 : %d\n", b )
}

func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保存 x 地址的值 */
   *x = *y      /* 将 y 赋值给 x */
   *y = temp    /* 将 temp 赋值给 y */
}
```

以上实例允许输出结果为：

```
交换前 a 的值 : 100
交换前 b 的值 : 200
交换后 a 的值 : 200
交换后 b 的值 : 100
```

Go 语言结构体

Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。

结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。

结构体表示一项记录，比如保存图书馆的书籍记录，每本书有以下属性：

- Title ：标题
- Author ： 作者
- Subject：学科
- ID：书籍ID

定义结构体

结构体定义需要使用 type 和 struct 语句。struct 语句定义一个新的数据类型，结构体有中一个或多个成员。type 语句设定了结构体的名称。结构体的格式如下：

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;
}
```

一旦定义了结构体类型，它就能用于变量的声明，语法格式如下：

```
variable_name := structure_variable_type {value1, value2...valuen}
```

访问结构体成员

如果要访问结构体成员，需要使用点号 (.) 操作符，格式为："结构体.成员名"。

结构体类型变量使用struct关键字定义，实例如下：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   fmt.Printf( "Book 1 title : %s\n", Book1.title)
   fmt.Printf( "Book 1 author : %s\n", Book1.author)
   fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
   fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)

   /* 打印 Book2 信息 */
   fmt.Printf( "Book 2 title : %s\n", Book2.title)
   fmt.Printf( "Book 2 author : %s\n", Book2.author)
   fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
   fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
}
```

以上实例执行运行结果为：

```
Book 1 title : Go 语言
Book 1 author : www.runoob.com
Book 1 subject : Go 语言教程
Book 1 book_id : 6495407
Book 2 title : Python 教程
Book 2 author : www.runoob.com
Book 2 subject : Python 语言教程
Book 2 book_id : 6495700
```

结构体作为函数参数

你可以向其他数据类型一样将结构体类型作为参数传递给函数。并以以上实例的方式访问结构体变量：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(Book1)

   /* 打印 Book2 信息 */
   printBook(Book2)
}

func printBook( book Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

以上实例执行运行结果为：

```
Book title : Go 语言
Book author : www.runoob.com
Book subject : Go 语言教程
Book book_id : 6495407
Book title : Python 教程
Book author : www.runoob.com
Book subject : Python 语言教程
Book book_id : 6495700
```

结构体指针

你可以定义指向结构体的指针类似于其他指针变量，格式如下：

```
var struct_pointer *Books
```

以上定义的指针变量可以存储结构体变量的地址。查看结构体变量地址，可以将 & 符号放置于结构体变量前：

```
struct_pointer = &Book1;
```

使用结构体指针访问结构体成员，使用 "." 操作符：

```
struct_pointer.title;
```

接下来让我们使用结构体指针重写以上实例，代码如下：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* Declare Book1 of type Book */
   var Book2 Books        /* Declare Book2 of type Book */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(&Book1)

   /* 打印 Book2 信息 */
   printBook(&Book2)
}
func printBook( book *Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

以上实例执行运行结果为：

```
Book title : Go 语言
Book author : www.runoob.com
Book subject : Go 语言教程
Book book_id : 6495407
Book title : Python 教程
Book author : www.runoob.com
Book subject : Python 语言教程
Book book_id : 6495700
```

Go 语言切片(Slice)

Go 语言切片是对数组的抽象。

Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组"),与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

定义切片

你可以声明一个未指定大小的数组来定义切片：

```
var identifier []type
```

切片不需要说明长度。

或使用make()函数来创建切片:

```
var slice1 []type = make([]type, len)

也可以简写为

slice1 := make([]type, len)
```

也可以指定容量，其中capacity为可选参数。

```
make([]T, length, capacity)
```

这里 len 是数组的长度并且也是切片的初始长度。

切片初始化

```
s :=[] int {1,2,3 }
```

直接初始化切片，[]表示是切片类型，{1,2,3}初始化值依次是1,2,3.其cap=len=3

```
s := arr[:]
```

初始化切片s,是数组arr的引用

```
s := arr[startIndex:endIndex]
```

将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片

```
s := arr[startIndex:]
```

缺省endIndex时将表示一直到arr的最后一个元素

```
s := arr[:endIndex]
```

缺省startIndex时将表示从arr的第一个元素开始

```
s1 := s[startIndex:endIndex]
```

通过切片s初始化切片s1

```
s :=make([]int,len,cap)
```

通过内置函数make()初始化切片s,[]int 标识为其元素类型为int的切片

len() 和 cap() 函数

切片是可索引的，并且可以由 len() 方法获取长度。

切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

以下为具体实例：

```
package main

import "fmt"

func main() {
   var numbers = make([]int,3,5)

   printSlice(numbers)
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

以上实例运行输出结果为:

```
len=3 cap=5 slice=[0 0 0]
```

空(nil)切片

一个切片在未初始化之前默认为 nil，长度为 0，实例如下：

```
package main

import "fmt"

func main() {
   var numbers []int

   printSlice(numbers)

   if(numbers == nil){
      fmt.Printf("切片是空的")
   }
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

以上实例运行输出结果为:

```
len=0 cap=0 slice=[]
切片是空的
```

切片截取

可以通过设置下限及上限来设置截取切片 *[lower-bound:upper-bound]*，实例如下：

```
package main

import "fmt"

func main() {
   /* 创建切片 */
   numbers := []int{0,1,2,3,4,5,6,7,8}
   printSlice(numbers)

   /* 打印原始切片 */
   fmt.Println("numbers ==", numbers)

   /* 打印子切片从索引1(包含) 到索引4(不包含)*/
   fmt.Println("numbers[1:4] ==", numbers[1:4])

   /* 默认下限为 0*/
   fmt.Println("numbers[:3] ==", numbers[:3])

   /* 默认上限为 len(s)*/
   fmt.Println("numbers[4:] ==", numbers[4:])

   numbers1 := make([]int,0,5)
   printSlice(numbers1)

   /* 打印子切片从索引  0(包含) 到索引 2(不包含) */
   number2 := numbers[:2]
   printSlice(number2)

   /* 打印子切片从索引 2(包含) 到索引 5(不包含) */
   number3 := numbers[2:5]
   printSlice(number3)

}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

执行以上代码输出结果为：

```
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=0 cap=5 slice=[]
len=2 cap=9 slice=[0 1]
len=3 cap=7 slice=[2 3 4]
```

append() 和 copy() 函数

如果想增加切片的容量，我们必须创建一个新的更大的切片并把原分片的内容都拷贝过来。

下面的代码描述了从拷贝切片的 copy 方法和向切片追加新元素的 append 方法。

```
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* 允许追加空切片 */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* 向切片添加一个元素 */
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* 同时添加多个元素 */
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* 创建切片 numbers1 是之前切片的两倍容量*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* 拷贝 numbers 的内容到 numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

以上代码执行输出结果为：

```
len=0 cap=0 slice=[]
len=1 cap=2 slice=[0]
len=2 cap=2 slice=[0 1]
len=5 cap=8 slice=[0 1 2 3 4]
len=5 cap=16 slice=[0 1 2 3 4]
```

Go 语言范围(Range)

Go 语言中 range 关键字用于for循环中迭代数组(array)、切片(slice)、链表(channel)或集合(map)的元素。在数组和切片中它返回元素的索引值，在集合中返回 key-value 对的 key 值。

实例

```
package main
import "fmt"
func main() {
    //这是我们使用range去求一个slice的和。使用数组跟这个很类似
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    //在数组上使用range将传入index和值两个变量。上面那个例子我们不需要使用该元素的序号，所以我们使用空白
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    //range也可以用在map的键值对上。
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    //range也可以用来枚举Unicode字符串。第一个参数是字符的索引，第二个是字符（Unicode的值）本身。
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

以上实例运行输出结果为：

```
sum: 9
index: 1
a -> apple
b -> banana
0 103
1 111
```

Go 语言Map(集合)

Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。

Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的。

定义 Map

可以使用内建函数 make 也可以使用 map 关键字来定义 Map:

```
/* 声明变量，默认 map 是 nil */
var map_variable map[key_data_type]value_data_type

/* 使用 make 函数 */
map_variable = make(map[key_data_type]value_data_type)
```

如果不初始化 map，那么就会创建一个 nil map。nil map 不能用来存放键值对

实例

下面实例演示了创建和使用map:

```
package main

import "fmt"

func main() {
   var countryCapitalMap map[string]string
   /* 创建集合 */
   countryCapitalMap = make(map[string]string)

   /* map 插入 key-value 对，各个国家对应的首都 */
   countryCapitalMap["France"] = "Paris"
   countryCapitalMap["Italy"] = "Rome"
   countryCapitalMap["Japan"] = "Tokyo"
   countryCapitalMap["India"] = "New Delhi"

   /* 使用 key 输出 map 值 */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }

   /* 查看元素在集合中是否存在 */
   captial, ok := countryCapitalMap["United States"]
   /* 如果 ok 是 true, 则存在，否则不存在 */
   if(ok){
      fmt.Println("Capital of United States is", captial)
   }else {
      fmt.Println("Capital of United States is not present")
   }
}
```

以上实例运行结果为：

```
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Capital of United States is not present
```

delete() 函数

delete() 函数用于删除集合的元素, 参数为 map 和其对应的 key。实例如下：

```
package main

import "fmt"

func main() {
   /* 创建 map */
   countryCapitalMap := map[string] string {"France":"Paris","Italy":"Rome","Japan":"Toky

   fmt.Println("原始 map")

   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }

   /* 删除元素 */
   delete(countryCapitalMap,"France");
   fmt.Println("Entry for France is deleted")

   fmt.Println("删除元素后 map")

   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
}
```

以上实例运行结果为：

```
原始 map
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Entry for France is deleted
删除元素后 map
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
```

Go 语言递归函数

递归，就是在运行的过程中调用自己。

语法格式如下：

```
func recursion() {
   recursion() /* 函数调用自身 */
}

func main() {
   recursion()
}
```

Go 语言支持递归。但我们在使用递归时，开发者需要设置退出条件，否则递归将陷入无限循环中。

递归函数对于解决数学上的问题是非常有用的，就像计算阶乘，生成斐波那契数列等。

阶乘

以下实例通过 Go 语言的递归函数实例阶乘：

```
package main

import "fmt"

func Factorial(x int) (result int) {
  if x == 0 {
    result = 1;
  } else {
    result = x * Factorial(x - 1);
  }
  return;
}

func main() {
    var i int = 15
    fmt.Printf("%d 的阶乘是 %d\n", i, Factorial(i))
}
```

以上实例执行输出结果为：

```
15 的阶乘是 1307674368000
```

斐波那契数列

以下实例通过 Go 语言的递归函数实现斐波那契数列：

```
package main

import "fmt"

func fibonaci(n int) int {
  if n < 2 {
   return n
  }
  return fibonaci(n-2) + fibonaci(n-1)
}

func main() {
    var i int
    for i = 0; i < 10; i++ {
       fmt.Printf("%d\t", fibonaci(i))
    }
}
```

以上实例执行输出结果为：

```
0    1    1    2    3    5    8    13    21    34
```

Go 语言类型转换

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。Go 语言类型转换基本格式如下：

```
type_name(expression)
```

type_name 为类型，expression 为表达式。

实例

以下实例中将整型转化为浮点型，并计算结果，将结果赋值给浮点型变量：

```
package main

import "fmt"

func main() {
   var sum int = 17
   var count int = 5
   var mean float32

   mean = float32(sum)/float32(count)
   fmt.Printf("mean 的值为: %f\n",mean)
}
```

以上实例执行输出结果为：

```
mean 的值为: 3.400000
```

Go 语言接口

Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。

实例

```
/* 定义接口 */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   method_name3 [return_type]
   ...
   method_namen [return_type]
}

/* 定义结构体 */
type struct_name struct {
   /* variables */
}

/* 实现接口方法 */
func (struct_name_variable struct_name) method_name1() [return_type] {
   /* 方法实现 */
}
...
func (struct_name_variable struct_name) method_namen() [return_type] {
   /* 方法实现*/
}
```

实例

```
package main

import (
    "fmt"
)

type Phone interface {
    call()
}

type NokiaPhone struct {
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}

type IPhone struct {
}

func (iPhone IPhone) call() {
    fmt.Println("I am iPhone, I can call you!")
}

func main() {
    var phone Phone

    phone = new(NokiaPhone)
    phone.call()

    phone = new(IPhone)
    phone.call()

}
```

在上面的例子中，我们定义了一个接口Phone，接口里面有一个方法call()。然后我们在main函数里面定义了一个Phone类型变量，并分别为之赋值为NokiaPhone和IPhone。然后调用call()方法，输出结果如下：

```
I am Nokia, I can call you!
I am iPhone, I can call you!
```

Go 错误处理

Go 语言通过内置的错误接口提供了非常简单的错误处理机制。

error类型是一个接口类型，这是它的定义：

```
type error interface {
    Error() string
}
```

我们可以在编码中通过实现 error 接口类型来生成错误信息。

函数通常在最后的返回值中返回错误信息。使用errors.New 可返回一个错误信息：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
    // 实现
}
```

在下面的例子中，我们在调用Sqrt的时候传递的一个负数，然后就得到了non-nil的error对象，将此对象与nil比较，结果为true，所以fmt.Println(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

```
result, err:= Sqrt(-1)

if err != nil {
   fmt.Println(err)
}
```

实例

```
package main

import (
    "fmt"
)

// 定义一个 DivideError 结构
type DivideError struct {
    dividee int
    divider int
}

// 实现      `error` 接口
func (de *DivideError) Error() string {
    strFormat := `
    Cannot proceed, the divider is zero.
    dividee: %d
    divider: 0
`
    return fmt.Sprintf(strFormat, de.dividee)
}

// 定义 `int` 类型除法运算的函数
func Divide(varDividee int, varDivider int) (result int, errorMsg string) {
    if varDivider == 0 {
        dData := DivideError{
            dividee: varDividee,
            divider: varDivider,
        }
        errorMsg = dData.Error()
        return
    } else {
        return varDividee / varDivider, ""
    }

}

func main() {

    // 正常情况
    if result, errorMsg := Divide(100, 10); errorMsg == "" {
        fmt.Println("100/10 = ", result)
    }
    // 当被除数为零的时候会返回错误信息
    if _, errorMsg := Divide(100, 0); errorMsg != "" {
        fmt.Println("errorMsg is: ", errorMsg)
    }

}
```

执行以上程序，输出结果为：

```
100/10 =  10
errorMsg is:
    Cannot proceed, the divider is zero.
    dividee: 100
    divider: 0
```

Go 语言开发工具

LiteIDE

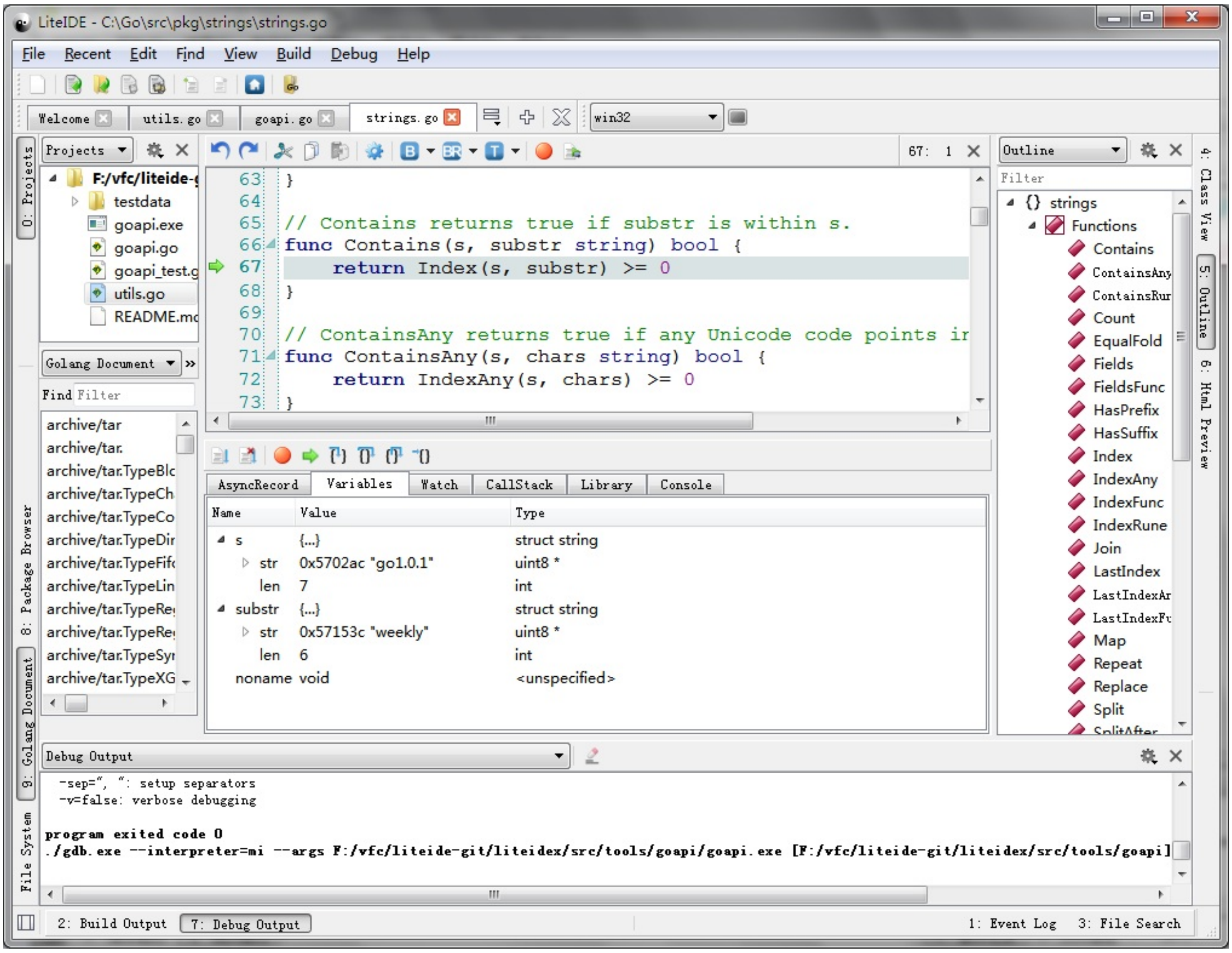
LiteIDE是一款开源、跨平台的轻量级Go语言集成开发环境（IDE）。

支持的操作系统

- Windows x86 (32-bit or 64-bit)
- Linux x86 (32-bit or 64-bit)

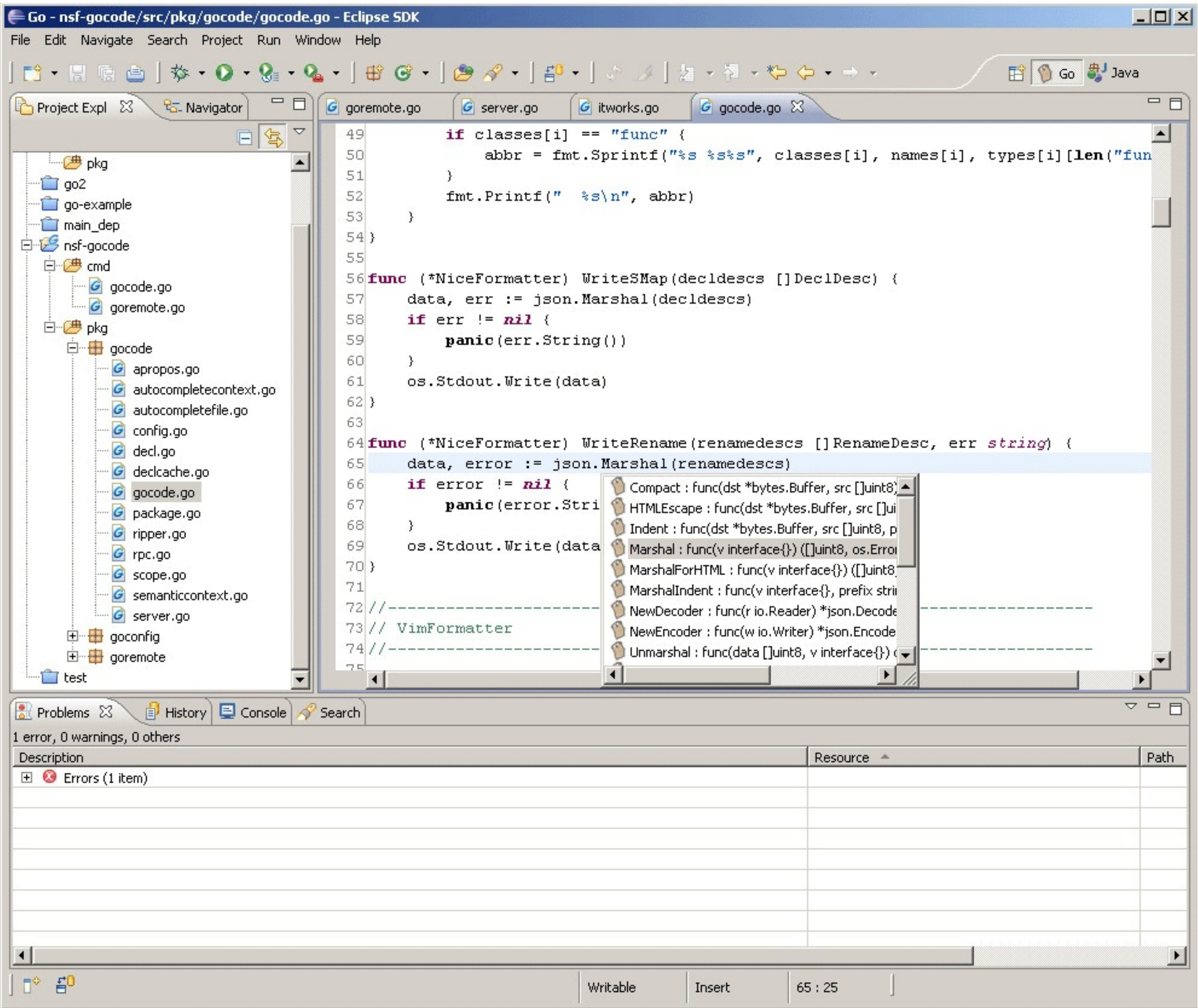
下载地址 ：http://sourceforge.net/projects/liteide/files/

源码地址 ：https://github.com/visualfc/liteide



Eclipse

Eclipse也是非常常用的开发利器，以下介绍如何使用Eclipse来编写Go程序。



Eclipse编辑Go的主界面

1. 首先下载并安装好Eclipse
2. 下载goclipse插件 http://code.google.com/p/goclipse/wiki/InstallationInstructions
3. 下载gocode，用于go的代码补全提示

   gocode的github地址：

   ```
   https://github.com/nsf/gocode
   ```

   在windows下要安装git，通常用msysgit

   再在cmd下安装：

   ```
   go get -u github.com/nsf/gocode
   ```

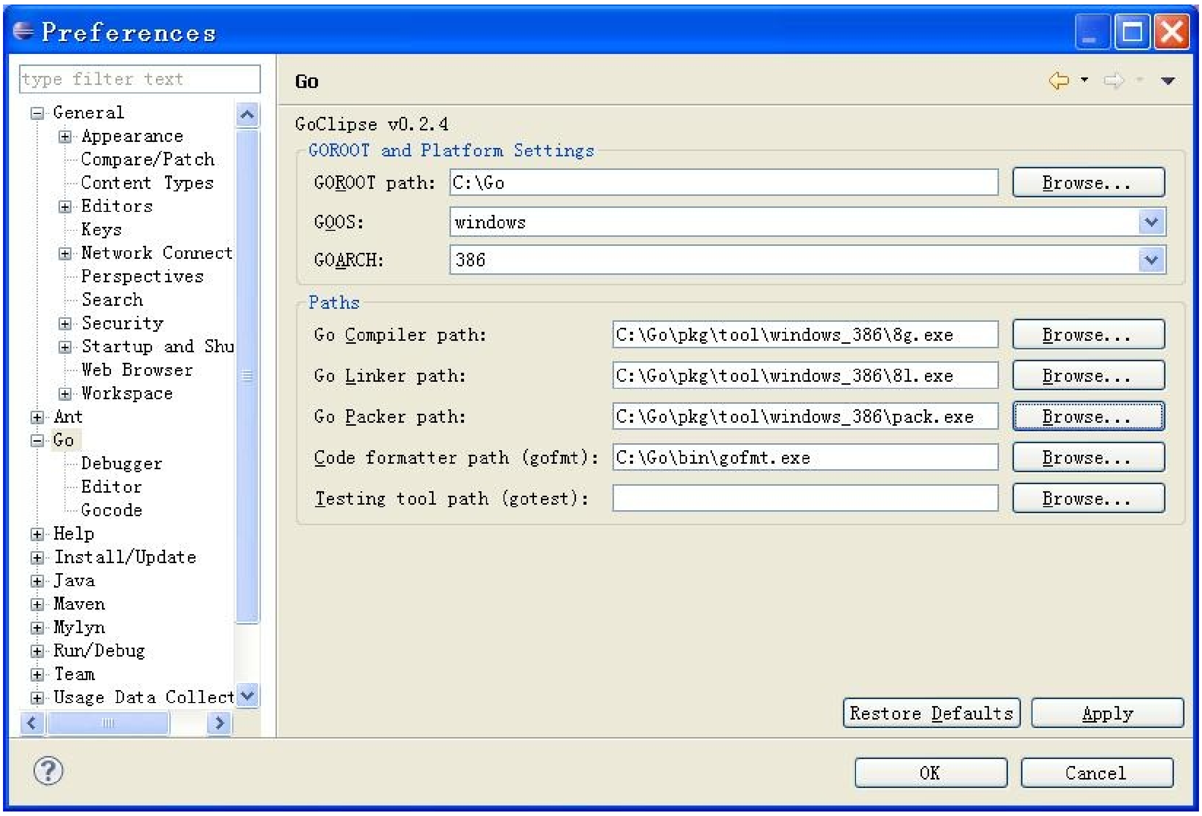
   也可以下载代码，直接用go build来编译，会生成gocode.exe

4. 下载MinGW并按要求装好

5. 配置插件

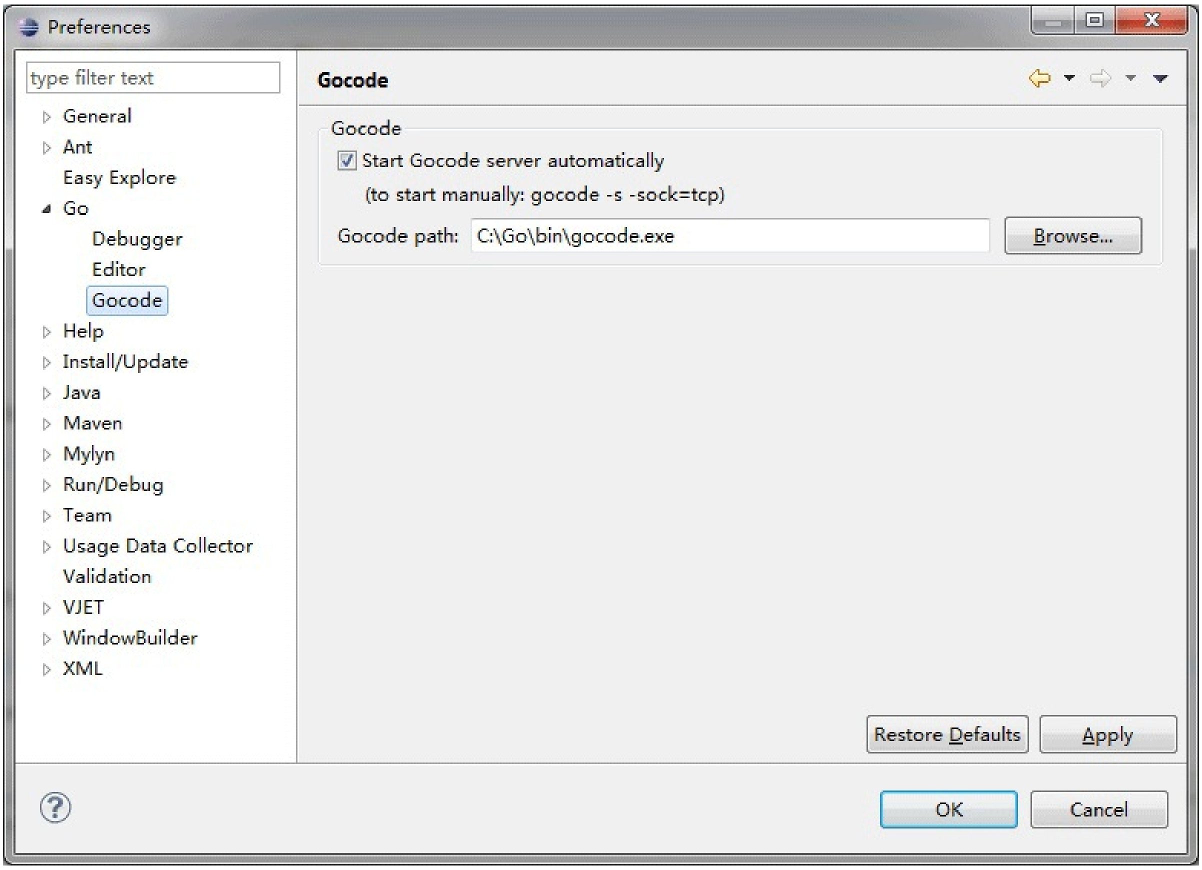
Windows->Reference->Go

(1).配置Go的编译器



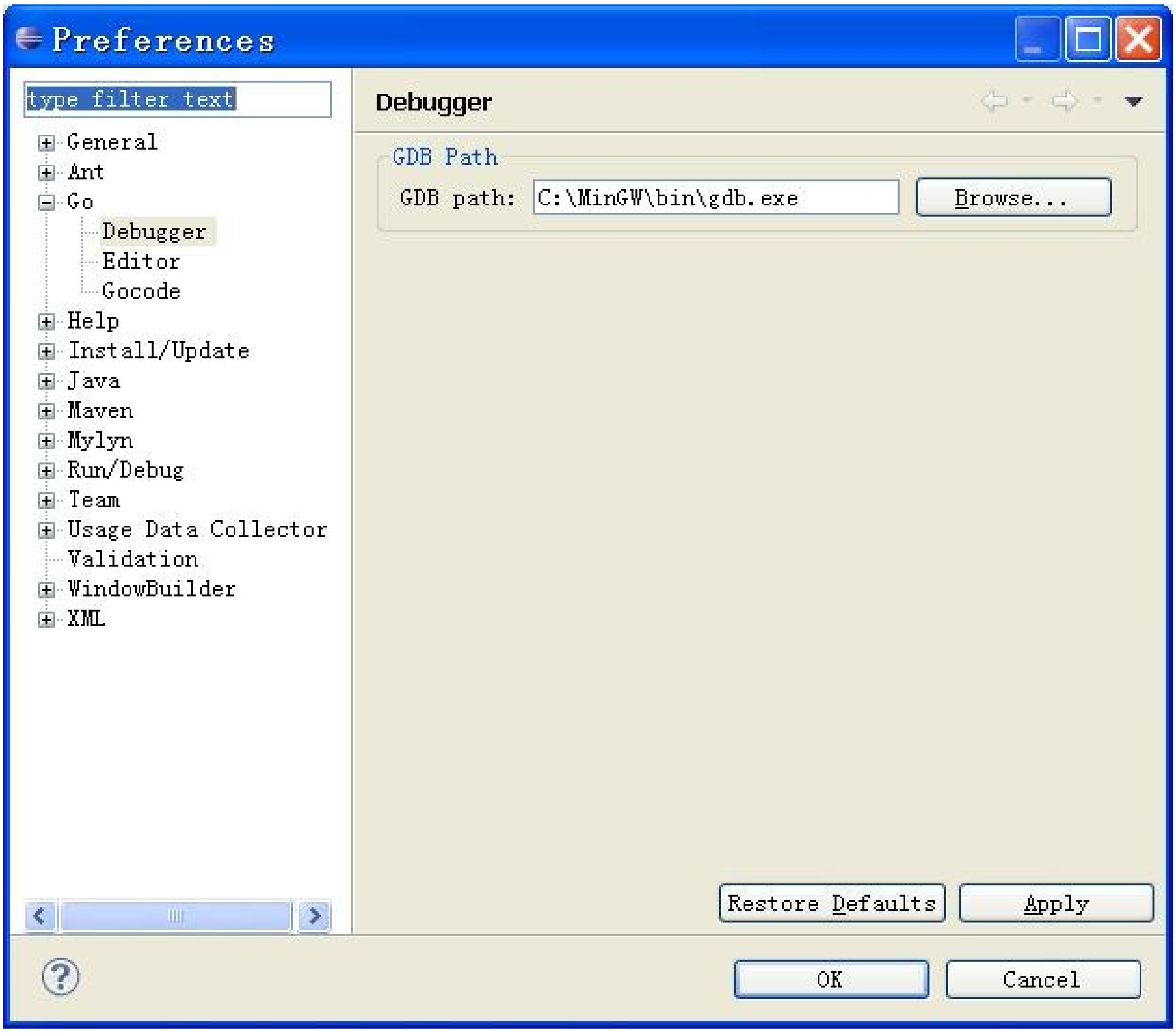
设置Go的一些基础信息

(2).配置Gocode（可选，代码补全），设置Gocode路径为之前生成的gocode.exe文件



设置gocode信息

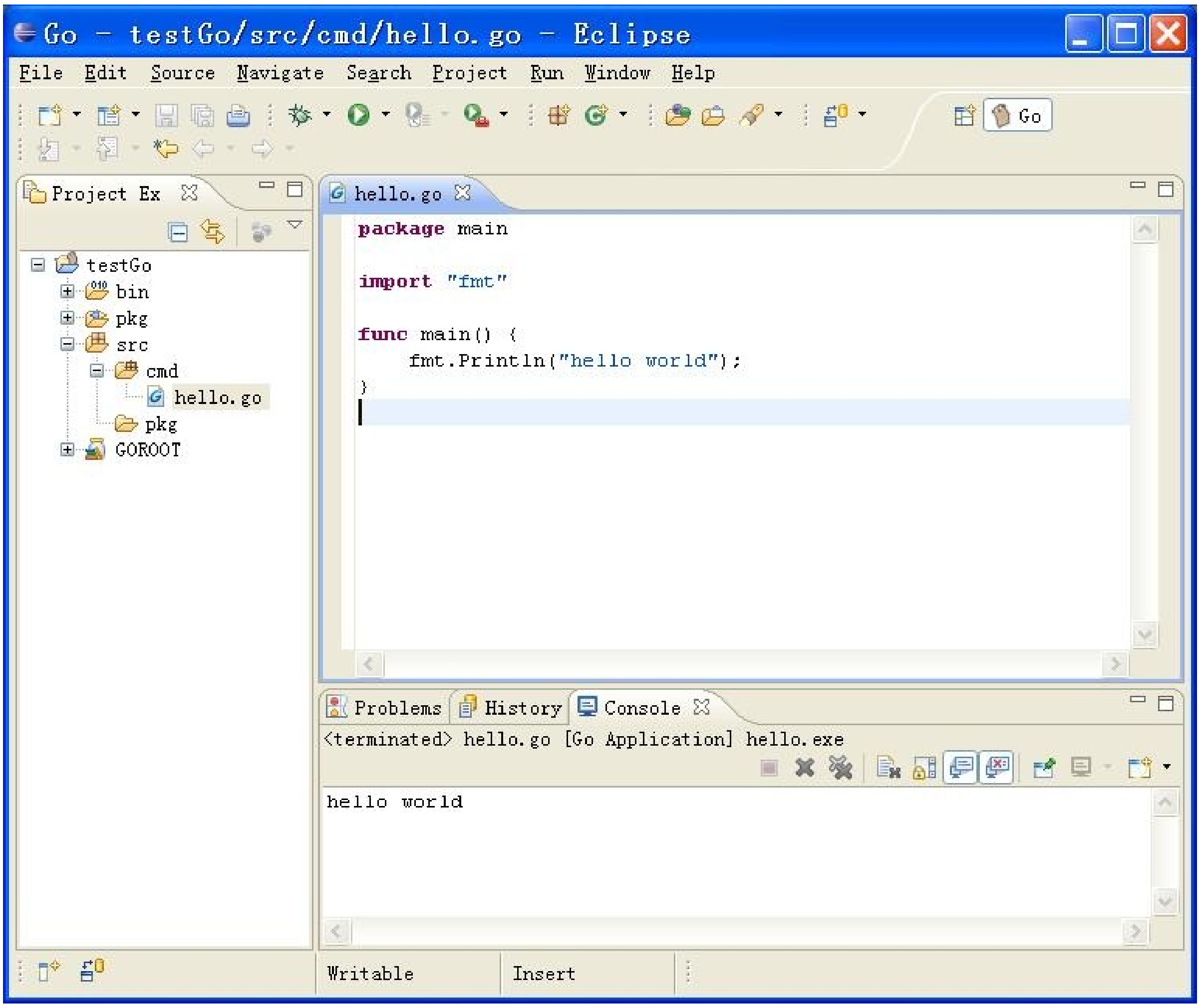
(3).配置GDB（可选，做调试用），设置GDB路径为MingW安装目录下的gdb.exe文件



设置GDB信息

6. 测试是否成功

新建一个go工程，再建立一个hello.go。如下图：



新建项目编辑文件

调试如下（要在console中用输入命令来调试）：

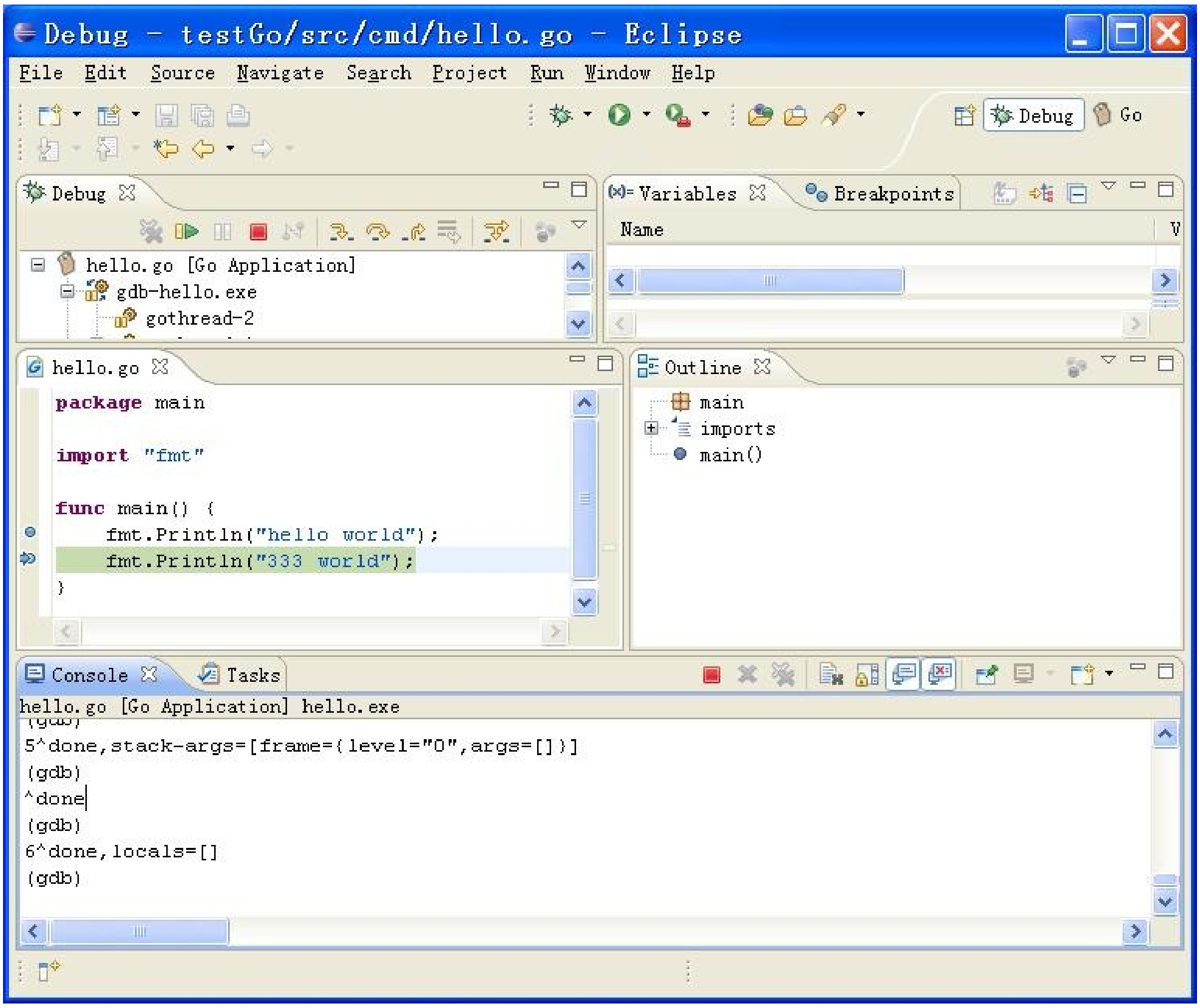

Debug - testGo/src/cmd/hello.go - Eclipse
File Edit Source Navigate Search Project Run Window Help
Debug
Go
Debug
Variables
Breakpoints
Name
hello.go [Go Application]
gdb-hello.exe
gothread-2
hello.go
package main
import "fmt"
func main() {
fmt.Println("hello world");
fmt.Println("333 world");
}
Outline
main
imports
main()
Console
Tasks
hello.go [Go Application] hello.exe
5^done,stack-args=[frame={level="0",args=[]}]
(gdb)
^done
(gdb)
6^done,locals=[]
(gdb)

图1.16 调试Go程序

W3School 设计模式教程

来源：设计模式教程

整理：飞龙

设计模式简介

设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

什么是 **GOF**（四人帮，全拼 **Gang of Four**）？

在 1994 年，由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 四人合著出版了一本名为 **Design Patterns - Elements of Reusable Object-Oriented Software**（中文译名：设计模式 **-** 可复用的面向对象软件元素） 的书，该书首次提到了软件开发中设计模式的概念。

四位作者合称 **GOF**（四人帮，全拼 **Gang of Four**）。他们所提出的设计模式主要是基于以下的面向对象设计原则。

- 对接口编程而不是对实现编程。
- 优先使用对象组合而不是继承。

设计模式的使用

设计模式在软件开发中的两个主要用途。

开发人员的共同平台

设计模式提供了一个标准的术语系统，且具体到特定的情景。例如，单例设计模式意味着使用单个对象，这样所有熟悉单例设计模式的开发人员都能使用单个对象，并且可以通过这种方式告诉对方，程序使用的是单例模式。

最佳的实践

设计模式已经经历了很长一段时间的发展，它们提供了软件开发过程中面临的一般问题的最佳解决方案。学习这些模式有助于经验不足的开发人员通过一种简单快捷的方式来学习软件设计。

设计模式的类型

根据设计模式的参考书 **Design Patterns - Elements of Reusable Object-Oriented Software**（中文译名：设计模式 **-** 可复用的面向对象软件元素） 中所提到的，总共有 23 种设计模式。这些模式可以分为三大类：创建型模式（Creational Patterns）、结构型模式（Structural Patterns）、行为型模式（Behavioral Patterns）。当然，我们还会讨论另一类设计模式：J2EE 设计模式。

模式 **&** 描述	包括
创建型模式 这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用新的运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。	工厂模式（Factory Pattern） 抽象工厂模式（Abstract Factory Pattern） 单例模式（Singleton Pattern） 建造者模式（Builder Pattern） 原型模式（Prototype Pattern）
结构型模式 这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。	适配器模式（Adapter Pattern） 桥接模式（Bridge Pattern） 过滤器模式（Filter、Criteria Pattern） 组合模式（Composite Pattern） 装饰器模式（Decorator Pattern） 外观模式（Facade Pattern） 享元模式（Flyweight Pattern） 代理模式（Proxy Pattern）
行为型模式 这些设计模式特别关注对象之间的通信。	责任链模式（Chain of Responsibility Pattern） 命令模式（Command Pattern） 解释器模式（Interpreter Pattern） 迭代器模式（Iterator Pattern） 中介者模式（Mediator Pattern） 备忘录模式（Memento Pattern） 观察者模式（Observer Pattern） 状态模式（State Pattern） 空对象模式（Null Object Pattern） 策略模式（Strategy Pattern） 模板模式（Template Pattern） 访问者模式（Visitor Pattern）
J2EE 模式 这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的。	MVC 模式（MVC Pattern） 业务代表模式（Business Delegate Pattern） 组合实体模式（Composite Entity Pattern） 数据访问对象模式（Data Access Object Pattern） 前端控制器模式（Front Controller Pattern） 拦截过滤器模式（Intercepting Filter Pattern） 服务定位器模式（Service Locator Pattern） 传输对象模式（Transfer Object Pattern）

下面用一个图片来整体描述一下设计模式之间的关系：

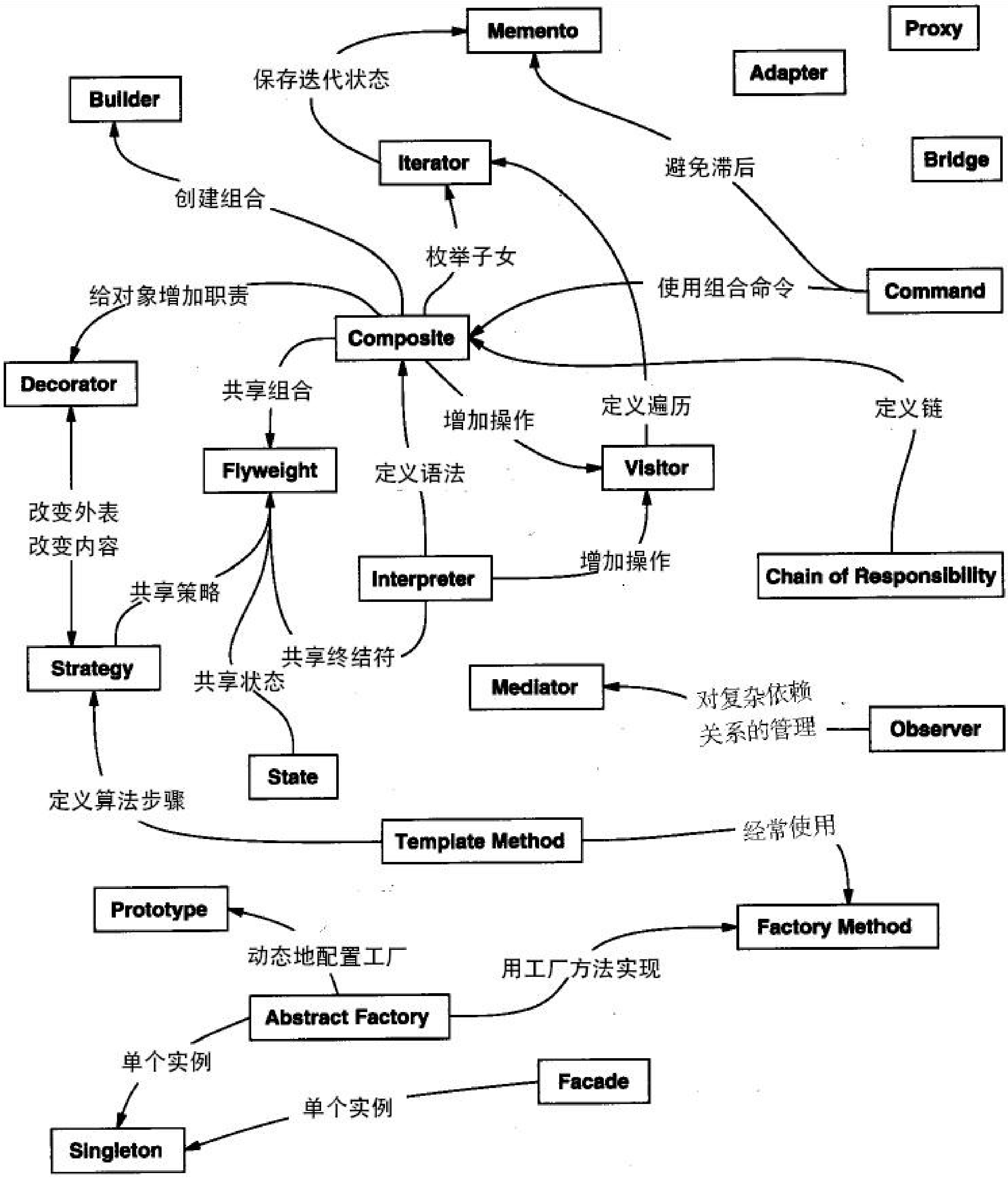


图　设计模式之间的关系

设计模式的六大原则

1、开闭原则（Open Close Principle）

开闭原则的意思是：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

2、里氏代换原则（**Liskov Substitution Principle**）

里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

3、依赖倒转原则（**Dependence Inversion Principle**）

这个原则是开闭原则的基础，具体内容：针对对接口编程，依赖于抽象而不依赖于具体。

4、接口隔离原则（**Interface Segregation Principle**）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

5、迪米特法则，又称最少知道原则（**Demeter Principle**）

最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

6、合成复用原则（**Composite Reuse Principle**）

合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承。

工厂模式

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

介绍

意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决接口选择的问题。

何时使用：我们明确地计划不同条件下创建不同实例时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码：创建过程在其子类执行。

应用实例： 1、您需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。 2、Hibernate 换数据库只需换方言和驱动就可以。

优点： 1、一个调用者想创建一个对象，只要知道其名称就可以了。 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。 3、屏蔽产品的具体实现，调用者只关心产品的接口。

缺点：每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

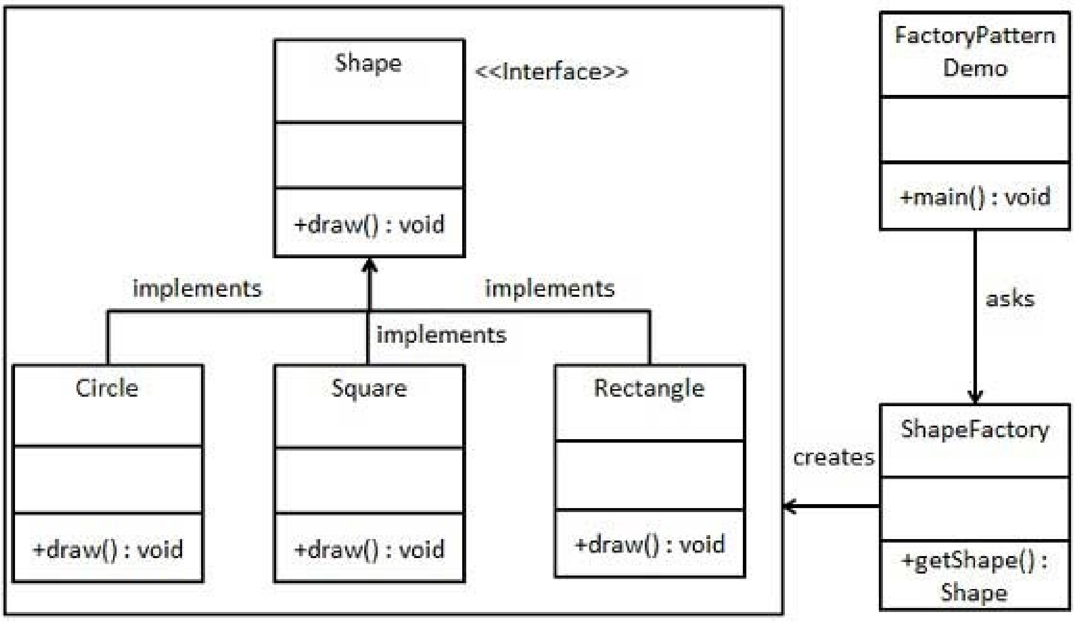
使用场景： 1、日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。 2、数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。 3、设计一个连接服务器的框架，需要三个协议，"POP3"、"IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口。

注意事项：作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

实现

我们将创建一个 *Shape* 接口和实现 *Shape* 接口的实体类。下一步是定义工厂类 *ShapeFactory*。

FactoryPatternDemo，我们的演示类使用 *ShapeFactory* 来获取 *Shape* 对象。它将向 *ShapeFactory* 传递信息（*CIRCLE / RECTANGLE / SQUARE*），以便获取它所需对象的类型。



步骤 1

创建一个接口。

Shape.java

```
public interface Shape {
   void draw();
}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

步骤 3

创建一个工厂，生成基于给定信息的实体类的对象。

ShapeFactory.java

```
public class ShapeFactory {

   //使用 getShape 方法获取形状类型的对象
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

步骤 4

使用该工厂，通过传递类型信息来获取实体类的对象。

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //获取 Circle 的对象，并调用它的 draw 方法
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //调用 Circle 的 draw 方法
      shape1.draw();

      //获取 Rectangle 的对象，并调用它的 draw 方法
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //调用 Rectangle 的 draw 方法
      shape2.draw();

      //获取 Square 的对象，并调用它的 draw 方法
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //调用 Square 的 draw 方法
      shape3.draw();
   }
}
```

步骤 5

验证输出。

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

抽象工厂模式

抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

介绍

意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

主要解决：主要解决接口选择的问题。

何时使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

如何解决：在一个产品族里面，定义多个产品。

关键代码：在一个工厂里聚合多个同类产品。

应用实例：工作了，为了参加一些聚会，肯定有两套或多套衣服吧，比如说有商务装（成套，一系列具体产品）、时尚装（成套，一系列具体产品），甚至对于一个家庭来说，可能有商务女装、商务男装、时尚女装、时尚男装，这些也都是成套的，即一系列具体产品。假设一种情况（现实中是不存在的，要不然，没法进入共产主义了，但有利于说明抽象工厂模式），在您的家中，某一个衣柜（具体工厂）只能存放某一种这样的衣服（成套，一系列具体产品），每次拿这种成套的衣服时也自然要从这个衣柜中取出了。用 OO 的思想去理解，所有的衣柜（具体工厂）都是衣柜类的（抽象工厂）某一个，而每一件成套的衣服又包括具体的上衣（某一具体产品），裤子（某一具体产品），这些具体的上衣其实也都是上衣（抽象产品），具体的裤子也都是裤子（另一个抽象产品）。

优点：当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

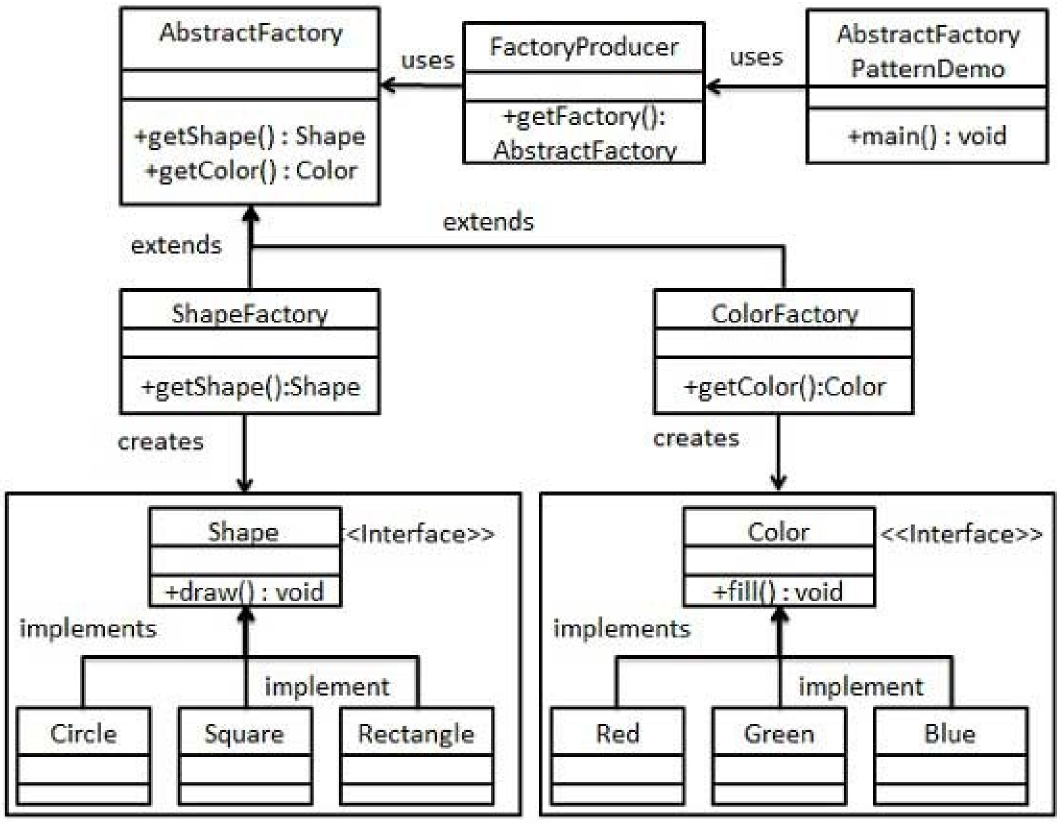
使用场景： 1、QQ 换皮肤，一整套一起换。 2、生成不同操作系统的程序。

注意事项：产品族难扩展，产品等级易扩展。

实现

我们将创建 *Shape* 和 *Color* 接口和实现这些接口的实体类。下一步是创建抽象工厂类 *AbstractFactory*。接着定义工厂类 *ShapeFactory* 和 *ColorFactory*，这两个工厂类都是扩展了 *AbstractFactory*。然后创建一个工厂创造器/生成器类 *FactoryProducer*。

AbstractFactoryPatternDemo，我们的演示类使用 *FactoryProducer* 来获取 *AbstractFactory* 对象。它将向 *AbstractFactory* 传递形状信息 *Shape*（*CIRCLE / RECTANGLE / SQUARE*），以便获取它所需对象的类型。同时它还向 *AbstractFactory* 传递颜色信息 *Color*（*RED / GREEN / BLUE*），以便获取它所需对象的类型。



步骤 1

为形状创建一个接口。

Shape.java

```
public interface Shape {
   void draw();
}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

步骤 3

为颜色创建一个接口。

Color.java

```
public interface Color {
   void fill();
}
```

步骤4

创建实现接口的实体类。

Red.java

```
public class Red implements Color {

   @Override
   public void fill() {
      System.out.println("Inside Red::fill() method.");
   }
}
```

Green.java

```
public class Green implements Color {

   @Override
   public void fill() {
      System.out.println("Inside Green::fill() method.");
   }
}
```

Blue.java

```
public class Blue implements Color {

   @Override
   public void fill() {
      System.out.println("Inside Blue::fill() method.");
   }
}
```

步骤 5

为 Color 和 Shape 对象创建抽象类来获取工厂。

AbstractFactory.java

```
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
```

步骤 6

创建扩展了 AbstractFactory 的工厂类，基于给定的信息生成实体类的对象。

ShapeFactory.java

```
public class ShapeFactory extends AbstractFactory {

   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }

   @Override
   Color getColor(String color) {
      return null;
   }
}
```

ColorFactory.java

```
public class ColorFactory extends AbstractFactory {

   @Override
   public Shape getShape(String shapeType){
      return null;
   }

   @Override
   Color getColor(String color) {
      if(color == null){
         return null;
      }
      if(color.equalsIgnoreCase("RED")){
         return new Red();
      } else if(color.equalsIgnoreCase("GREEN")){
         return new Green();
      } else if(color.equalsIgnoreCase("BLUE")){
         return new Blue();
      }
      return null;
   }
}
```

步骤 7

创建一个工厂创造器/生成器类，通过传递形状或颜色信息来获取工厂。

FactoryProducer.java

```
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();
      } else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }
      return null;
   }
}
```

步骤 8

使用 FactoryProducer 来获取 AbstractFactory，通过传递类型信息来获取实体类的对象。

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {

      //获取形状工厂
      AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

      //获取形状为 Circle 的对象
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //调用 Circle 的 draw 方法
      shape1.draw();

      //获取形状为 Rectangle 的对象
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //调用 Rectangle 的 draw 方法
      shape2.draw();

      //获取形状为 Square 的对象
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //调用 Square 的 draw 方法
      shape3.draw();

      //获取颜色工厂
      AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

      //获取颜色为 Red 的对象
      Color color1 = colorFactory.getColor("RED");

      //调用 Red 的 fill 方法
      color1.fill();

      //获取颜色为 Green 的对象
      Color color2 = colorFactory.getColor("Green");

      //调用 Green 的 fill 方法
      color2.fill();

      //获取颜色为 Blue 的对象
      Color color3 = colorFactory.getColor("BLUE");

      //调用 Blue 的 fill 方法
      color3.fill();
   }
}
```

步骤 9

验证输出。

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside Red::fill() method.
Inside Green::fill() method.
Inside Blue::fill() method.
```

单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

介绍

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

何时使用：当您想控制实例数目，节省系统资源的时候。

如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

关键代码：构造函数是私有的。

应用实例： 1、一个党只能有一个主席。 2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。 3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点： 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。 2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

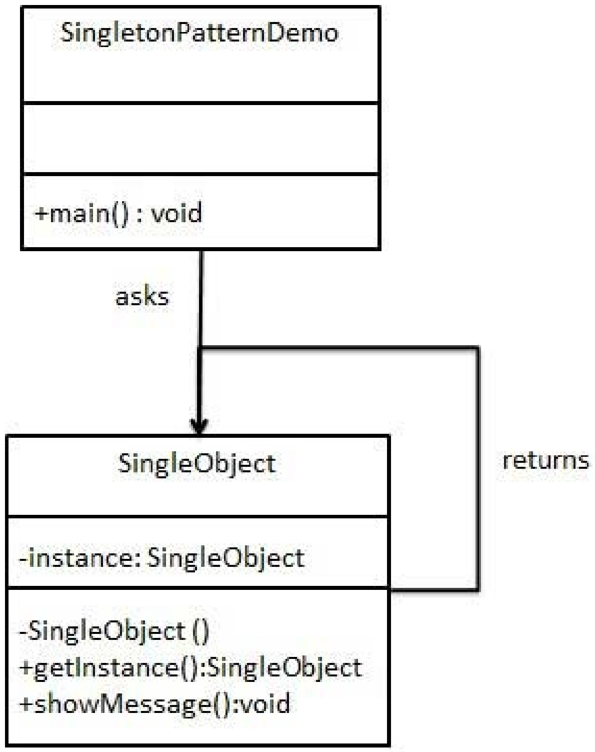
使用场景： 1、要求生产唯一序列号。 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

实现

我们将创建一个 *SingleObject* 类。*SingleObject* 类有它的私有构造函数和本身的一个静态实例。

SingleObject 类提供了一个静态方法，供外界获取它的静态实例。*SingletonPatternDemo*，我们的演示类使用 *SingleObject* 类来获取 *SingleObject* 对象。



步骤 1

创建一个 Singleton 类。

SingleObject.java

```
public class SingleObject {

   //创建 SingleObject 的一个对象
   private static SingleObject instance = new SingleObject();

   //让构造函数为 private，这样该类就不会被实例化
   private SingleObject(){}

   //获取唯一可用的对象
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

步骤 2

从 singleton 类获取唯一的对象。

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //不合法的构造函数
      //编译时错误：构造函数 SingleObject() 是不可见的
      //SingleObject object = new SingleObject();

      //获取唯一可用的对象
      SingleObject object = SingleObject.getInstance();

      //显示消息
      object.showMessage();
   }
}
```

步骤 3

验证输出。

```
Hello World!
```

单例模式的几种实现方式

单例模式的实现有多种方式，如下所示：

1、懒汉式，线程不安全

是否 **Lazy** 初始化：是

是否多线程安全：否

实现难度：易

描述：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。
这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

代码实例：

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

接下来介绍的几种实现方式都支持多线程，但是在性能上有所差异。

2、懒汉式，线程安全

是否 **Lazy** 初始化：是

是否多线程安全：是

实现难度：易

描述：这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。
优点：第一次调用才初始化，避免内存浪费。
缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。
getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。

代码实例：

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

3、饿汉式

是否 **Lazy** 初始化：否

是否多线程安全：是

实现难度：易

描述：这种方式比较常用，但容易产生垃圾对象。
优点：没有加锁，执行效率会提高。
缺点：类加载时就初始化，浪费内存。

它基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

代码实例：

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
    return instance;
    }
}
```

4、双检锁/双重校验锁（DCL，即 double-checked locking）

JDK 版本：JDK1.5 起

是否 **Lazy** 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。
getInstance() 的性能对应用程序很关键。

代码实例：

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
    }
    return singleton;
    }
}
```

5、登记式/静态内部类

是否 **Lazy** 初始化：是

是否多线程安全：是

实现难度：一般

描述：这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。
这种方式同样利用了 classloder 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有显示通过调用 getInstance 方法时，才会显示装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。

代码实例：

```
public class Singleton {
    private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
    return SingletonHolder.INSTANCE;
    }
}
```

6、枚举

JDK 版本：JDK1.5 起

是否 **Lazy** 初始化：否

是否多线程安全：是

实现难度：易

描述：这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。
这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象，绝对防止多次实例化。不过，由于 JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。
不能通过 reflection attack 来调用私有构造方法。

代码实例：

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {
    }
}
```

经验之谈：一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

建造者模式

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

介绍

意图：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

主要解决：主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

何时使用：一些基本部件不会变，而其组合经常变化的时候。

如何解决：将变与不变分离开。

关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

应用实例： 1、去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。 2、JAVA 中的 StringBuilder。

优点： 1、建造者独立，易扩展。 2、便于控制细节风险。

缺点： 1、产品必须有共同点，范围有限制。 2、如内部变化复杂，会有很多的建造类。

使用场景： 1、需要生成的对象具有复杂的内部结构。 2、需要生成的对象内部属性本身相互依赖。

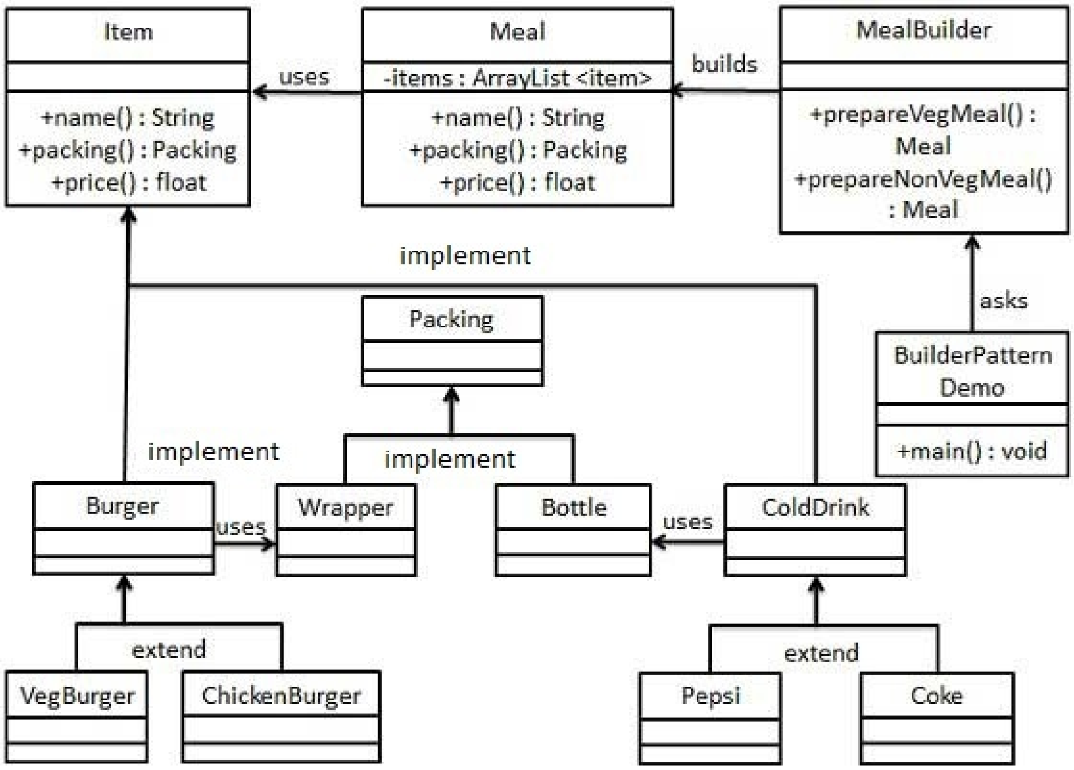
注意事项：与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。

实现

我们假设一个快餐店的商业案例，其中，一个典型的套餐可以是一个汉堡（Burger）和一杯冷饮（Cold drink）。汉堡（Burger）可以是素食汉堡（Veg Burger）或鸡肉汉堡（Chicken Burger），它们是包在纸盒中。冷饮（Cold drink）可以是可口可乐（coke）或百事可乐（pepsi），它们是装在瓶子中。

我们将创建一个表示食物条目（比如汉堡和冷饮）的 *Item* 接口和实现 *Item* 接口的实体类，以及一个表示食物包装的 *Packing* 接口和实现 *Packing* 接口的实体类，汉堡是包在纸盒中，冷饮是装在瓶子中。

然后我们创建一个 *Meal* 类，带有 *Item* 的 *ArrayList* 和一个通过结合 *Item* 来创建不同类型的 *Meal* 对象的 *MealBuilder*。*BuilderPatternDemo*，我们的演示类使用 *MealBuilder* 来创建一个 *Meal*。



步骤 1

创建一个表示食物条目和食物包装的接口。

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```
public interface Packing {
   public String pack();
}
```

步骤 2

创建实现 Packing 接口的实体类。

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

步骤 3

创建实现 Item 接口的抽象类，该类提供了默认的功能。

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

    @Override
    public Packing packing() {
       return new Bottle();
    }

    @Override
    public abstract float price();
}
```

步骤 4

创建扩展了 Burger 和 ColdDrink 的实体类。

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

步骤 5

创建一个 Meal 类，带有上面定义的 Item 对象。

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;
      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){
      for (Item item : items) {
         System.out.print("Item : "+item.name());
         System.out.print(", Packing : "+item.packing().pack());
         System.out.println(", Price : "+item.price());
      }
   }
}
```

步骤 6

创建一个 MealBuilder 类，实际的 builder 类负责创建 Meal 对象。

MealBuilder.java

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

步骤 7

BuiderPatternDemo 使用 MealBuider 来演示建造者模式（Builder Pattern）。

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {
      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " +vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " +nonVegMeal.getCost());
   }
}
```

步骤 8

验证输出。

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0

Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

原型模式

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

介绍

意图：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

主要解决：在运行期建立和删除原型。

何时使用： 1、当一个系统应该独立于它的产品创建，构成和表示时。 2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。 3、为了避免创建一个与产品类层次平行的工厂类层次时。 4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

如何解决：利用已有的一个原型对象，快速地生成和原型对象一样的实例。

关键代码： 1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()，在 .NET 中可以使用 Object 类的 MemberwiseClone() 方法来实现对象的浅拷贝或通过序列化的方式来实现深拷贝。 2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有稳定的接口。

应用实例： 1、细胞分裂。 2、JAVA 中的 Object clone() 方法。

优点： 1、性能提高。 2、逃避构造函数的约束。

缺点： 1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。 2、必须实现 Cloneable 接口。 3、逃避构造函数的约束。

使用场景： 1、资源优化场景。 2、类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。 3、性能和安全要求的场景。 4、通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。 5、一个对象多个修改者的场景。 6、一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多

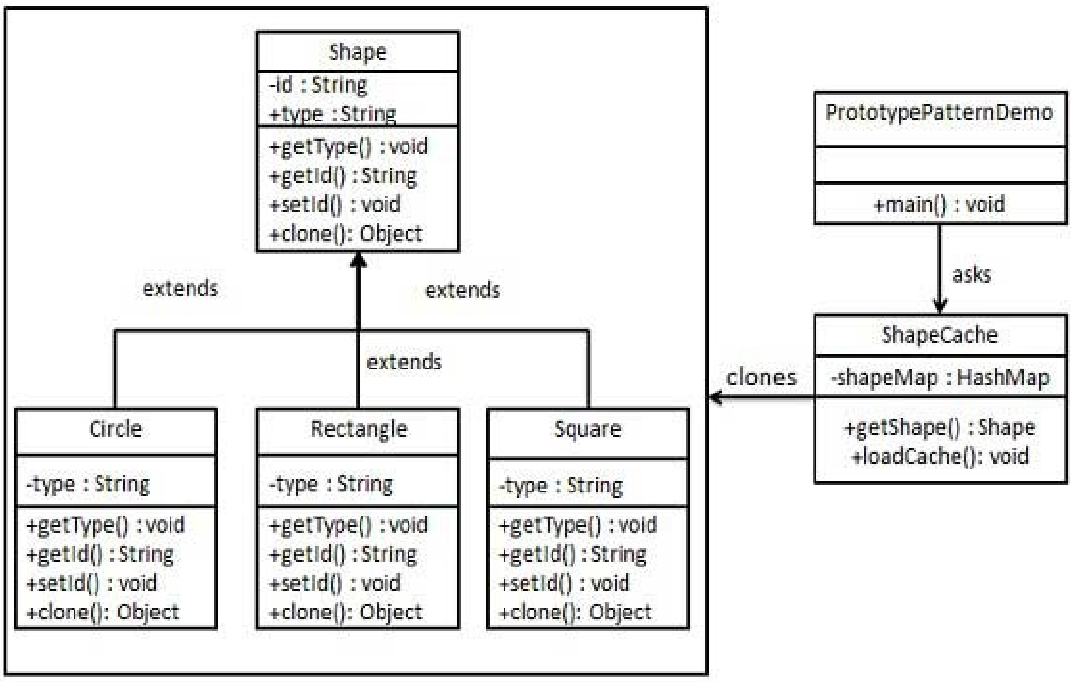
个对象供调用者使用。 7、在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与 Java 融为浑然一体，大家可以随手拿来使用。

注意事项：与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

实现

我们将创建一个抽象类 *Shape* 和扩展了 *Shape* 类的实体类。下一步是定义类 *ShapeCache*，该类把 shape 对象存储在一个 *Hashtable* 中，并在请求的时候返回它们的克隆。

PrototypPatternDemo，我们的演示类使用 *ShapeCache* 类来获取 *Shape* 对象。



步骤 1

创建一个实现了 *Clonable* 接口的抽象类。

Shape.java

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;
      try {
         clone = super.clone();
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }
      return clone;
   }
}
```

步骤 2

创建扩展了上面抽象类的实体类。

Rectangle.java

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

步骤 3

创建一个类，从数据库获取实体类，并把它们存储在一个 *Hashtable* 中。

ShapeCache.java

```
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap
      = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // 对每种形状都运行数据库查询，并创建该形状
   // shapeMap.put(shapeKey, shape);
   // 例如，我们要添加三种形状
   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(),rectangle);
   }
}
```

步骤 4

PrototypePatternDemo 使用 *ShapeCache* 类来获取存储在 *Hashtable* 中的形状的克隆。

PrototypePatternDemo.java

```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

步骤 5

验证输出。

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

适配器模式

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。

我们通过下面的实例来演示适配器模式的使用。其中，音频播放器设备只能播放 mp3 文件，通过使用一个更高级的音频播放器来播放 vlc 和 mp4 文件。

介绍

意图：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

主要解决：主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。

何时使用： 1、系统需要使用现有的类，而此类的接口不符合系统的需要。 2、想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。 3、通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

如何解决：继承或依赖（推荐）。

关键代码：适配器继承或依赖已有的对象，实现想要的目标接口。

应用实例： 1、美国电器 110V，中国 220V，就要有一个适配器将 110V 转化为 220V。 2、JAVA JDK 1.1 提供了 Enumeration 接口，而在 1.2 中提供了 Iterator 接口，想要使用 1.2 的 JDK，则要将以前系统的 Enumeration 接口转化为 Iterator 接口，这时就需要适配器模式。 3、在 LINUX 上运行 WINDOWS 程序。 4、JAVA 中的 jdbc。

优点： 1、可以让任何两个没有关联的类一起运行。 2、提高了类的复用。 3、增加了类的透明度。 4、灵活性好。

缺点： 1、过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。 2.由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

使用场景：有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式。

注意事项：适配器不是在详细设计时添加的，而是解决正在服役的项目的问题。

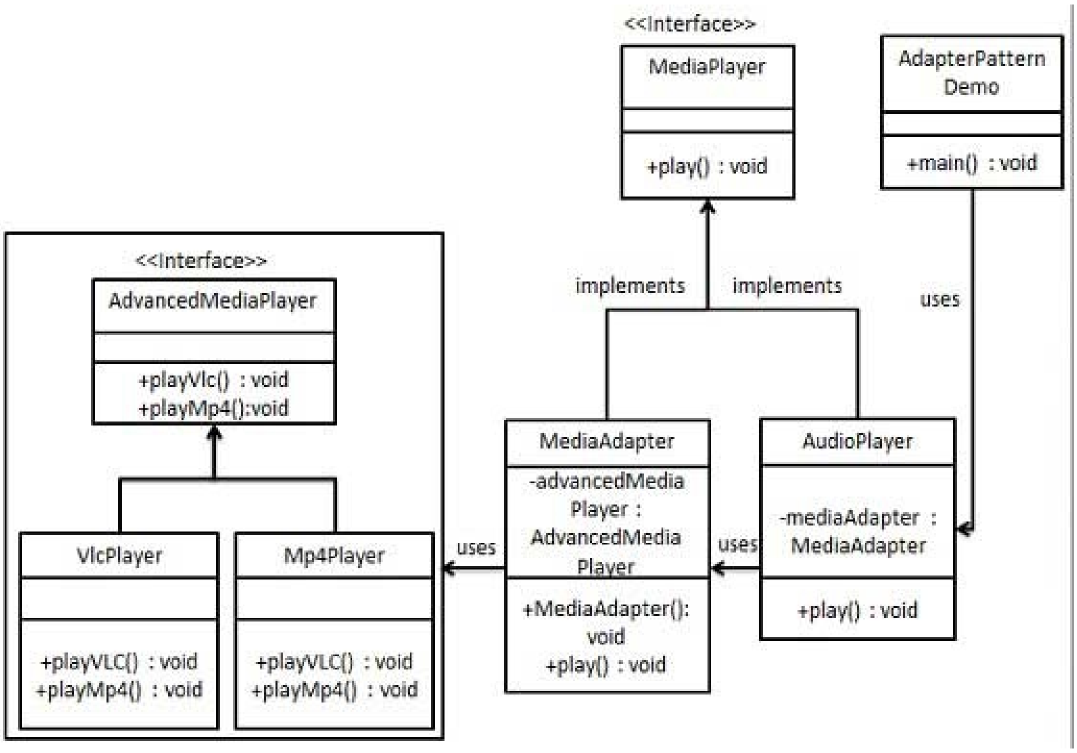
实现

我们有一个 *MediaPlayer* 接口和一个实现了 *MediaPlayer* 接口的实体类 *AudioPlayer*。默认情况下，*AudioPlayer* 可以播放 mp3 格式的音频文件。

我们还有另一个接口 *AdvancedMediaPlayer* 和实现了 *AdvancedMediaPlayer* 接口的实体类。该类可以播放 vlc 和 mp4 格式的文件。

我们想要让 *AudioPlayer* 播放其他格式的音频文件。为了实现这个功能，我们需要创建一个实现了 *MediaPlayer* 接口的适配器类 *MediaAdapter*，并使用 *AdvancedMediaPlayer* 对象来播放所需的格式。

AudioPlayer 使用适配器类 *MediaAdapter* 传递所需的音频类型，不需要知道能播放所需格式音频的实际类。*AdapterPatternDemo*，我们的演示类使用 *AudioPlayer* 类来播放各种格式。



步骤 1

为媒体播放器和更高级的媒体播放器创建接口。

MediaPlayer.java

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

步骤 2

创建实现了 *AdvancedMediaPlayer* 接口的实体类。

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //什么也不做
   }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //什么也不做
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

步骤 3

创建实现了 *MediaPlayer* 接口的适配器类。

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      } else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

步骤 4

创建实现了 *MediaPlayer* 接口的实体类。

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //播放 mp3 音乐文件的内置支持
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: "+ fileName);
      }
      //mediaAdapter 提供了播放其他文件格式的支持
      else if(audioType.equalsIgnoreCase("vlc")
         || audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }
      else{
         System.out.println("Invalid media. "+
            audioType + " format not supported");
      }
   }
}
```

步骤 5

使用 AudioPlayer 来播放不同类型的音频格式。

AdapterPatternDemo.java

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

步骤 6

验证输出。

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

桥接模式

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

我们通过下面的实例来演示桥接模式（Bridge Pattern）的用法。其中，可以使用相同的抽象类方法但是不同的桥接实现类，来画出不同颜色的圆。

介绍

意图：将抽象部分与实现部分分离，使它们都可以独立的变化。

主要解决：在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。

何时使用：实现系统可能有多个角度分类，每一种角度都可能变化。

如何解决：把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。

关键代码：抽象类依赖实现类。

应用实例： 1、猪八戒从天蓬元帅转世投胎到猪，转世投胎的机制将尘世划分为两个等级，即：灵魂和肉体，前者相当于抽象化，后者相当于实现化。生灵通过功能的委派，调用肉体对象的功能，使得生灵可以动态地选择。 2、墙上的开关，可以看到的开关是抽象的，不用管里面具体怎么实现的。

优点： 1、抽象和实现的分离。 2、优秀的扩展能力。 3、实现细节对客户透明。

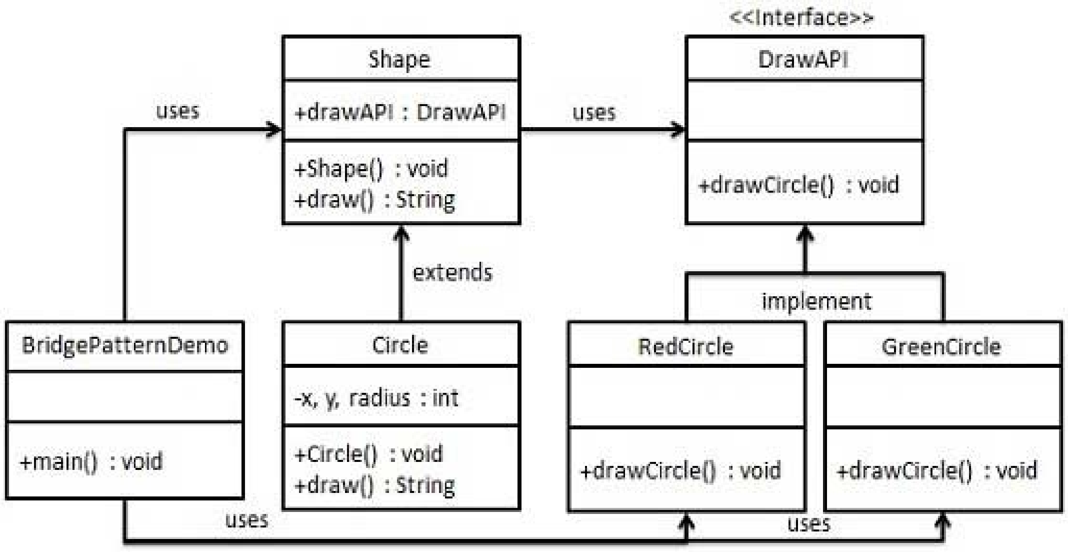
缺点：桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

使用场景： 1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。 2、对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。 3、一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

注意事项：对于两个独立变化的维度，使用桥接模式再适合不过了。

实现

我们有一个作为桥接实现的 *DrawAPI* 接口和实现了 *DrawAPI* 接口的实体类 *RedCircle*、*GreenCircle*。*Shape* 是一个抽象类，将使用 *DrawAPI* 的对象。*BridgePatternDemo*，我们的演示类使用 *Shape* 类来画出不同颜色的圆。



步骤 1

创建桥接实现接口。

DrawAPI.java

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

步骤 2

创建实现了 *DrawAPI* 接口的实体桥接实现类。

RedCircle.java

```
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: "
         + radius +", x: " +x+", "+ y +"]");
   }
}
```

GreenCircle.java

```
public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: "
         + radius +", x: " +x+", "+ y +"]");
   }
}
```

步骤 3

使用 *DrawAPI* 接口创建抽象类 *Shape*。

Shape.java

```
public abstract class Shape {
   protected DrawAPI drawAPI;
   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

步骤 4

创建实现了 *Shape* 接口的实体类。

Circle.java

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

步骤 5

使用 *Shape* 和 *DrawAPI* 类画出不同颜色的圆。

BridgePatternDemo.java

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

步骤 6

验证输出。

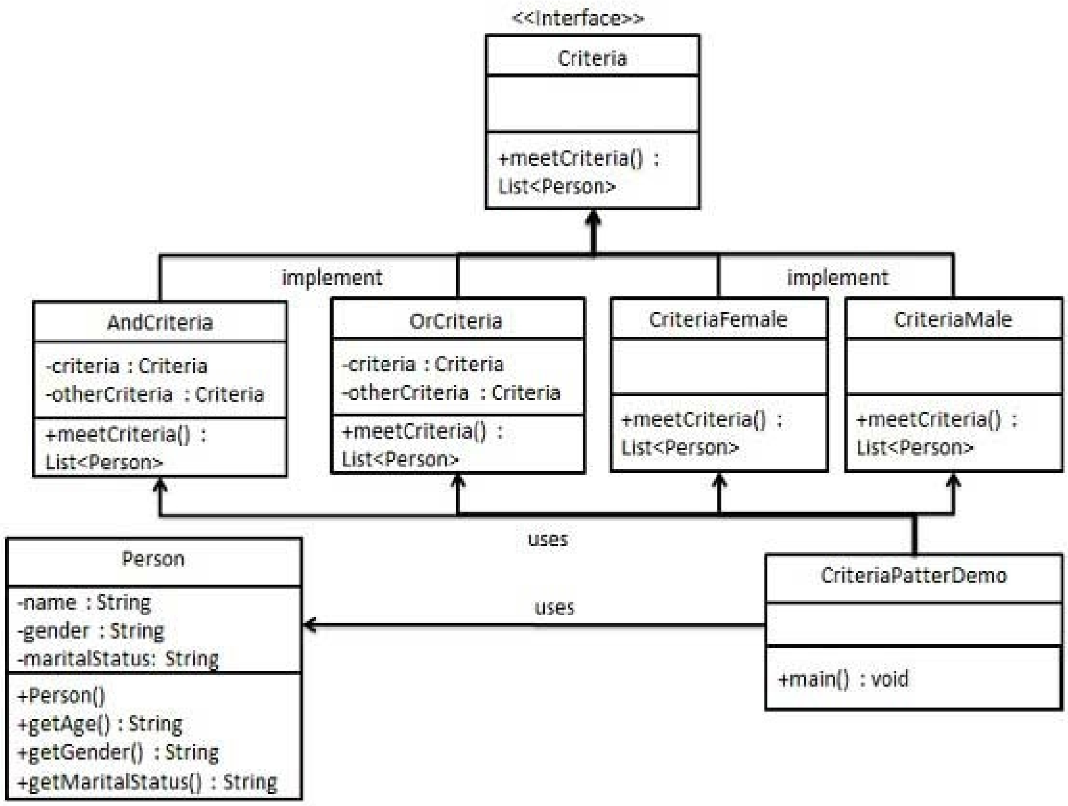
```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[  color: green, radius: 10, x: 100, 100]
```

过滤器模式

过滤器模式（Filter Pattern）或标准模式（Criteria Pattern）是一种设计模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准。

实现

我们将创建一个 *Person* 对象、*Criteria* 接口和实现了该接口的实体类，来过滤 *Person* 对象的列表。*CriteriaPatternDemo*，我们的演示类使用 *Criteria* 对象，基于各种标准和它们的结合来过滤 *Person* 对象的列表。



步骤 1

创建一个类，在该类上应用标准。

Person.java

```
public class Person {

   private String name;
   private String gender;
   private String maritalStatus;

   public Person(String name,String gender,String maritalStatus){
      this.name = name;
      this.gender = gender;
      this.maritalStatus = maritalStatus;
   }

   public String getName() {
      return name;
   }
   public String getGender() {
      return gender;
   }
   public String getMaritalStatus() {
      return maritalStatus;
   }
}
```

步骤 2

为标准（Criteria）创建一个接口。

Criteria.java

```
import java.util.List;

public interface Criteria {
   public List<Person> meetCriteria(List<Person> persons);
}
```

步骤 3

创建实现了 *Criteria* 接口的实体类。

CriteriaMale.java

```
import java.util.ArrayList;
import java.util.List;

public class CriteriaMale implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> malePersons = new ArrayList<Person>();
      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("MALE")){
            malePersons.add(person);
         }
      }
      return malePersons;
   }
}
```

CriteriaFemale.java

```
import java.util.ArrayList;
import java.util.List;

public class CriteriaFemale implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> femalePersons = new ArrayList<Person>();
      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("FEMALE")){
            femalePersons.add(person);
         }
      }
      return femalePersons;
   }
}
```

CriteriaSingle.java

```
import java.util.ArrayList;
import java.util.List;

public class CriteriaSingle implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> singlePersons = new ArrayList<Person>();
      for (Person person : persons) {
         if(person.getMaritalStatus().equalsIgnoreCase("SINGLE")){
            singlePersons.add(person);
         }
      }
      return singlePersons;
   }
}
```

AndCriteria.java

```
import java.util.List;

public class AndCriteria implements Criteria {

   private Criteria criteria;
   private Criteria otherCriteria;

   public AndCriteria(Criteria criteria, Criteria otherCriteria) {
      this.criteria = criteria;
      this.otherCriteria = otherCriteria;
   }

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> firstCriteriaPersons = criteria.meetCriteria(persons);
      return otherCriteria.meetCriteria(firstCriteriaPersons);
   }
}
```

OrCriteria.java

```
import java.util.List;

public class OrCriteria implements Criteria {

   private Criteria criteria;
   private Criteria otherCriteria;

   public OrCriteria(Criteria criteria, Criteria otherCriteria) {
      this.criteria = criteria;
      this.otherCriteria = otherCriteria;
   }

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> firstCriteriaItems = criteria.meetCriteria(persons);
      List<Person> otherCriteriaItems = otherCriteria.meetCriteria(persons);

      for (Person person : otherCriteriaItems) {
         if(!firstCriteriaItems.contains(person)){
            firstCriteriaItems.add(person);
         }
      }
      return firstCriteriaItems;
   }
}
```

步骤4

使用不同的标准（Criteria）和它们的结合来过滤 *Person* 对象的列表。

CriteriaPatternDemo.java

```
public class CriteriaPatternDemo {
   public static void main(String[] args) {
      List<Person> persons = new ArrayList<Person>();

      persons.add(new Person("Robert","Male", "Single"));
      persons.add(new Person("John","Male", "Married"));
      persons.add(new Person("Laura","Female", "Married"));
      persons.add(new Person("Diana","Female", "Single"));
      persons.add(new Person("Mike","Male", "Single"));
      persons.add(new Person("Bobby","Male", "Single"));

      Criteria male = new CriteriaMale();
      Criteria female = new CriteriaFemale();
      Criteria single = new CriteriaSingle();
      Criteria singleMale = new AndCriteria(single, male);
      Criteria singleOrFemale = new OrCriteria(single, female);

      System.out.println("Males: ");
      printPersons(male.meetCriteria(persons));

      System.out.println("\nFemales: ");
      printPersons(female.meetCriteria(persons));

      System.out.println("\nSingle Males: ");
      printPersons(singleMale.meetCriteria(persons));

      System.out.println("\nSingle Or Females: ");
      printPersons(singleOrFemale.meetCriteria(persons));
   }

   public static void printPersons(List<Person> persons){
      for (Person person : persons) {
         System.out.println("Person : [ Name : " + person.getName()
            +", Gender : " + person.getGender()
            +", Marital Status : " + person.getMaritalStatus()
            +" ]");
      }
   }
}
```

步骤 5

验证输出。

```
Males:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : John, Gender : Male, Marital Status : Married ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Females:
Person : [ Name : Laura, Gender : Female, Marital Status : Married ]
Person : [ Name : Diana, Gender : Female, Marital Status : Single ]

Single Males:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Single Or Females:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : Diana, Gender : Female, Marital Status : Single ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]
Person : [ Name : Laura, Gender : Female, Marital Status : Married ]
```

组合模式

组合模式（Composite Pattern），又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。

我们通过下面的实例来演示组合模式的用法。实例演示了一个组织中员工的层次结构。

介绍

意图：将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

主要解决：它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

何时使用： 1、您想表示对象的部分-整体层次结构（树形结构）。 2、您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

如何解决：树枝和叶子实现统一接口，树枝内部组合该接口。

关键代码：树枝内部组合该接口，并且含有内部属性 List，里面放 Component。

应用实例： 1、算术表达式包括操作数、操作符和另一个操作数，其中，另一个操作符也可以是操作树、操作符和另一个操作数。 2、在 JAVA AWT 和 SWING 中，对于 Button 和 Checkbox 是树叶，Container 是树枝。

优点： 1、高层模块调用简单。 2、节点自由增加。

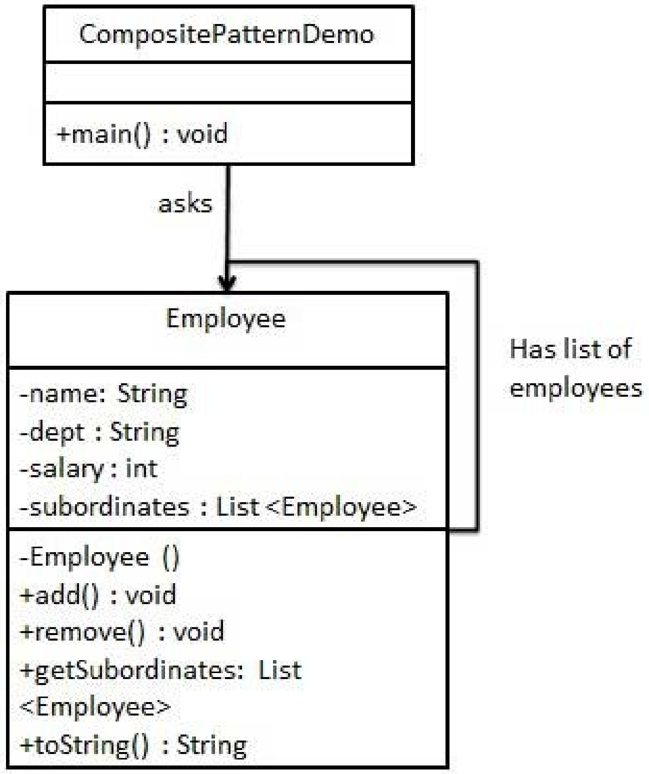
缺点：在使用组合模式时，其叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则。

使用场景：部分、整体场景，如树形菜单，文件、文件夹的管理。

注意事项：定义时为具体类。

实现

我们有一个类 *Employee*，该类被当作组合模型类。*CompositePatternDemo*，我们的演示类使用 *Employee* 类来添加部门层次结构，并打印所有员工。



步骤 1

创建 *Employee* 类，该类带有 *Employee* 对象的列表。

Employee.java

```
import java.util.ArrayList;
import java.util.List;

public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;

   //构造函数
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }

   public void remove(Employee e) {
      subordinates.remove(e);
   }

   public List<Employee> getSubordinates(){
     return subordinates;
   }

   public String toString(){
      return ("Employee :[ Name : "+ name
      +", dept : "+ dept + ", salary :"
      + salary+" ]");
   }
}
```

步骤 2

使用 *Employee* 类来创建和打印员工的层次结构。

CompositePatternDemo.java

```
public class CompositePatternDemo {
   public static void main(String[] args) {
      Employee CEO = new Employee("John","CEO", 30000);

      Employee headSales = new Employee("Robert","Head Sales", 20000);

      Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

      Employee clerk1 = new Employee("Laura","Marketing", 10000);
      Employee clerk2 = new Employee("Bob","Marketing", 10000);

      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
      Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

      CEO.add(headSales);
      CEO.add(headMarketing);

      headSales.add(salesExecutive1);
      headSales.add(salesExecutive2);

      headMarketing.add(clerk1);
      headMarketing.add(clerk2);

      //打印该组织的所有员工
      System.out.println(CEO);
      for (Employee headEmployee : CEO.getSubordinates()) {
         System.out.println(headEmployee);
         for (Employee employee : headEmployee.getSubordinates()) {
            System.out.println(employee);
         }
      }
   }
}
```

步骤 3

验证输出。

```
Employee :[ Name : John, dept : CEO, salary :30000 ]
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]
Employee :[ Name : Richard, dept : Sales, salary :10000 ]
Employee :[ Name : Rob, dept : Sales, salary :10000 ]
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]
Employee :[ Name : Bob, dept : Marketing, salary :10000 ]
```

装饰器模式

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

我们通过下面的实例来演示装饰器模式的用法。其中，我们将把一个形状装饰上不同的颜色，同时又不改变形状类。

介绍

意图：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

主要解决：一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

何时使用：在不想增加很多子类的情况下扩展类。

如何解决：将具体功能职责划分，同时继承装饰者模式。

关键代码： 1、Component 类充当抽象角色，不应该具体实现。 2、修饰类引用和继承 Component 类，具体扩展类重写父类方法。

应用实例： 1、孙悟空有 72 变，当他变成"庙宇"后，他的根本还是一只猴子，但是他又有了庙宇的功能。 2、不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

缺点：多层装饰比较复杂。

使用场景： 1、扩展一个类的功能。 2、动态增加功能，动态撤销。

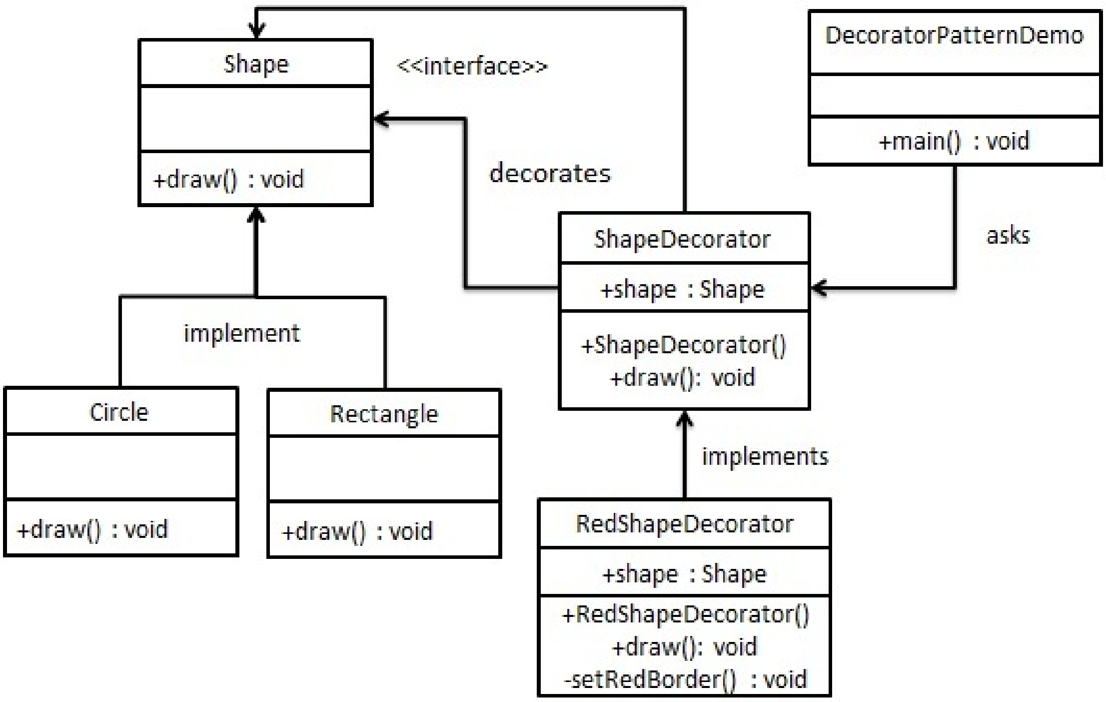
注意事项：可代替继承。

实现

我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类。然后我们创建一个实现了 *Shape* 接口的抽象装饰类 *ShapeDecorator*，并把 *Shape* 对象作为它的实例变量。

RedShapeDecorator 是实现了 *ShapeDecorator* 的实体类。

DecoratorPatternDemo，我们的演示类使用 *RedShapeDecorator* 来装饰 *Shape* 对象。



步骤 1

创建一个接口。

Shape.java

```
public interface Shape {
   void draw();
}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

步骤 3

创建实现了 *Shape* 接口的抽象装饰类。

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

步骤 4

创建扩展了 *ShapeDecorator* 类的实体装饰类。

RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

步骤 5

使用 *RedShapeDecorator* 来装饰 *Shape* 对象。

DecoratorPatternDemo.java

```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

步骤 6

验证输出。

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

外观模式

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。

这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

介绍

意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。

何时使用： 1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。 2、定义系统的入口。

如何解决：客户端不与系统耦合，外观类与系统耦合。

关键代码：在客户端和复杂系统之间再加一层，这一次将调用顺序、依赖关系等处理好。

应用实例： 1、去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。 2、JAVA 的三层开发模式。

优点： 1、减少系统相互依赖。 2、提高灵活性。 3、提高了安全性。

缺点：不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。

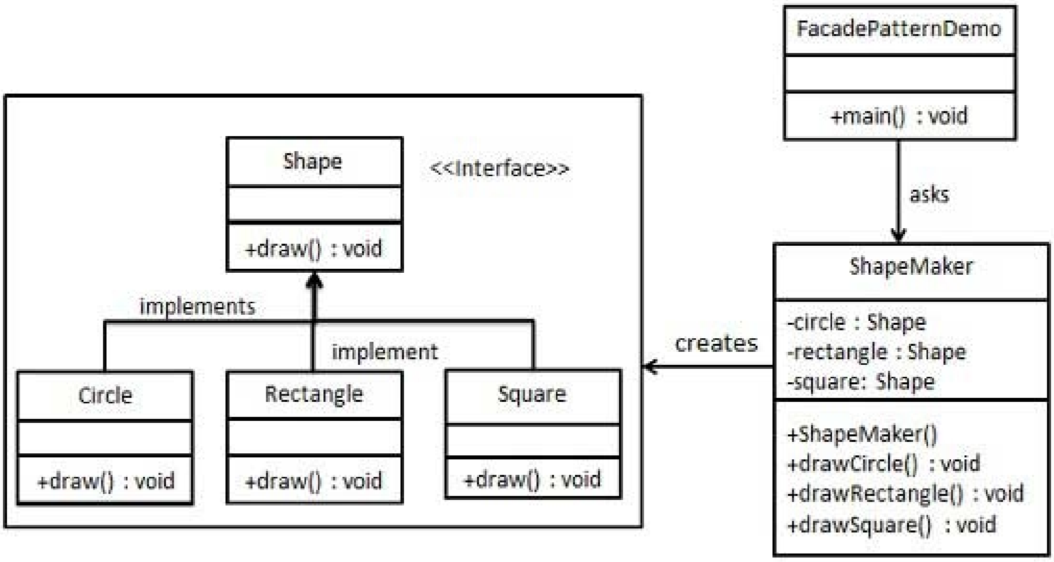
使用场景： 1、为复杂的模块或子系统提供外界访问的模块。 2、子系统相对独立。 3、预防低水平人员带来的风险。

注意事项：在层次化结构中，可以使用外观模式定义系统中每一层的入口。

实现

我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类。下一步是定义一个外观类 *ShapeMaker*。

ShapeMaker 类使用实体类来代表用户对这些类的调用。*FacadePatternDemo*，我们的演示类使用 *ShapeMaker* 类来显示结果。



步骤 1

创建一个接口。

Shape.java

```
public interface Shape {
   void draw();
}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Rectangle::draw()");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Square::draw()");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Circle::draw()");
   }
}
```

步骤 3

创建一个外观类。

ShapeMaker.java

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

步骤 4

使用该外观类画出各种类型的形状。

FacadePatternDemo.java

```
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

步骤 5

验证输出。

```
Circle::draw()
Rectangle::draw()
Square::draw()
```

享元模式

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。我们将通过创建 5 个对象来画出 20 个分布于不同位置的圆来演示这种模式。由于只有 5 种可用的颜色，所以 color 属性被用来检查现有的 *Circle* 对象。

介绍

意图：运用共享技术有效地支持大量细粒度的对象。

主要解决：在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。

何时使用： 1、系统中有大量对象。 2、这些对象消耗大量内存。 3、这些对象的状态大部分可以外部化。 4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。 5、系统不依赖于这些对象身份，这些对象是不可分辨的。

如何解决：用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。

关键代码：用 HashMap 存储这些对象。

应用实例： 1、JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面。 2、数据库的数据池。

优点：大大减少对象的创建，降低系统的内存，使效率提高。

缺点：提高了系统的负责度，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱。

使用场景： 1、系统有大量相似对象。 2、需要缓冲池的场景。

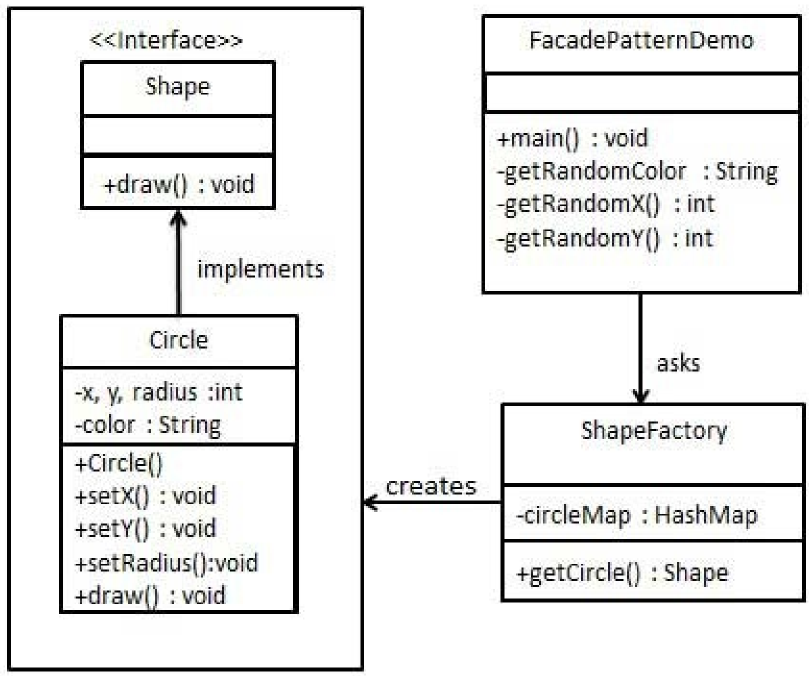
注意事项： 1、注意划分外部状态和内部状态，否则可能会引起线程安全问题。 2、这些类必须有一个工厂对象加以控制。

实现

我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类 *Circle*。下一步是定义工厂类 *ShapeFactory*。

ShapeFactory 有一个 *Circle* 的 *HashMap*，其中键名为 *Circle* 对象的颜色。无论何时接收到请求，都会创建一个特定颜色的圆。*ShapeFactory* 检查它的 *HashMap* 中的 circle 对象，如果找到 *Circle* 对象，则返回该对象，否则将创建一个存储在 hashmap 中以备后续使用的新对象，并把该对象返回到客户端。

FlyWeightPatternDemo，我们的演示类使用 *ShapeFactory* 来获取 *Shape* 对象。它将向 *ShapeFactory* 传递信息（*red / green / blue/ black / white*），以便获取它所需对象的颜色。



步骤 1

创建一个接口。

Shape.java

```
public interface Shape {
   void draw();
}
```

步骤 2

创建实现接口的实体类。

Circle.java

```
public class Circle implements Shape {
   private String color;
   private int x;
   private int y;
   private int radius;

   public Circle(String color){
      this.color = color;
   }

   public void setX(int x) {
      this.x = x;
   }

   public void setY(int y) {
      this.y = y;
   }

   public void setRadius(int radius) {
      this.radius = radius;
   }

   @Override
   public void draw() {
      System.out.println("Circle: Draw() [Color : " + color
         +", x : " + x +", y :" + y +", radius :" + radius);
   }
}
```

步骤 3

创建一个工厂，生成基于给定信息的实体类的对象。

ShapeFactory.java

```
import java.util.HashMap;

public class ShapeFactory {
   private static final HashMap<String, Shape> circleMap = new HashMap();

   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " + color);
      }
      return circle;
   }
}
```

步骤 4

使用该工厂，通过传递颜色信息来获取实体类的对象。

FlyweightPatternDemo.java

```
public class FlyweightPatternDemo {
   private static final String colors[] =
      { "Red", "Green", "Blue", "White", "Black" };
   public static void main(String[] args) {

      for(int i=0; i < 20; ++i) {
         Circle circle =
            (Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

步骤 5

验证输出。

```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

代理模式

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。

在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。

介绍

意图：为其他对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

何时使用：想在访问一个类时做一些控制。

如何解决：增加中间层。

关键代码：实现与被代理类组合。

应用实例： 1、Windows 里面的快捷方式。 2、猪八戒去找高翠兰结果是孙悟空变的，可以这样理解：把高翠兰的外貌抽象出来，高翠兰本人和孙悟空都实现了这个接口，猪八戒访问高翠兰的时候看不出来这个是孙悟空，所以说孙悟空是高翠兰代理类。 3、买火车票不一定在火车站买，也可以去代售点。 4、一张支票或银行存单是账户中资金的代理。支票在市场交易中用来代替现金，并提供对签发人账号上资金的控制。 5、spring aop。

优点： 1、职责清晰。 2、高扩展性。 3、智能化。

缺点： 1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。 2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

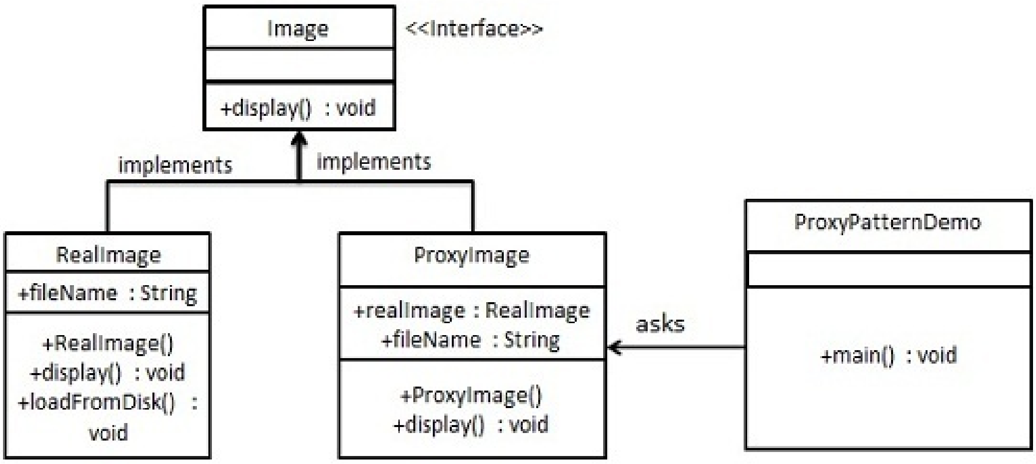
使用场景：按职责来划分，通常有以下使用场景： 1、远程代理。 2、虚拟代理。 3、Copy-on-Write 代理。 4、保护（Protect or Access）代理。 5、Cache代理。 6、防火墙（Firewall）代理。 7、同步化（Synchronization）代理。 8、智能引用（Smart Reference）代理。

注意事项： 1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。 2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

实现

我们将创建一个 *Image* 接口和实现了 *Image* 接口的实体类。*ProxyImage* 是一个代理类，减少 *RealImage* 对象加载的内存占用。

ProxyPatternDemo，我们的演示类使用 *ProxyImage* 来获取要加载的 *Image* 对象，并按照需求进行显示。



步骤 1

创建一个接口。

Image.java

```
public interface Image {
   void display();
}
```

步骤 2

创建实现接口的实体类。

RealImage.java

```
public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}
```

ProxyImage.java

```
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

步骤 3

当被请求时，使用 *ProxyImage* 来获取 *RealImage* 类的对象。

ProxyPatternDemo.java

```
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      //图像将从磁盘加载
      image.display();
      System.out.println("");
      //图像将无法从磁盘加载
      image.display();
   }
}
```

步骤 4

验证输出。

```
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
```

责任链模式

顾名思义，责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

介绍

意图：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。

主要解决：职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了。

何时使用：在处理消息的时候以过滤很多道。

如何解决：拦截的类都实现统一接口。

关键代码：Handler 里面聚合它自己，在 HanleRequest 里判断是否合适，如果没达到条件则向下传递，向谁传递之前 set 进去。

应用实例： 1、红楼梦中的"击鼓传花"。 2、JS 中的事件冒泡。 3、JAVA WEB 中 Apache Tomcat 对 Encoding 的处理，Struts2 的拦截器，jsp servlet 的 Filter。

优点： 1、降低耦合度。它将请求的发送者和接收者解耦。 2、简化了对象。使得对象不需要知道链的结构。 3、增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任。 4、增加新的请求处理类很方便。

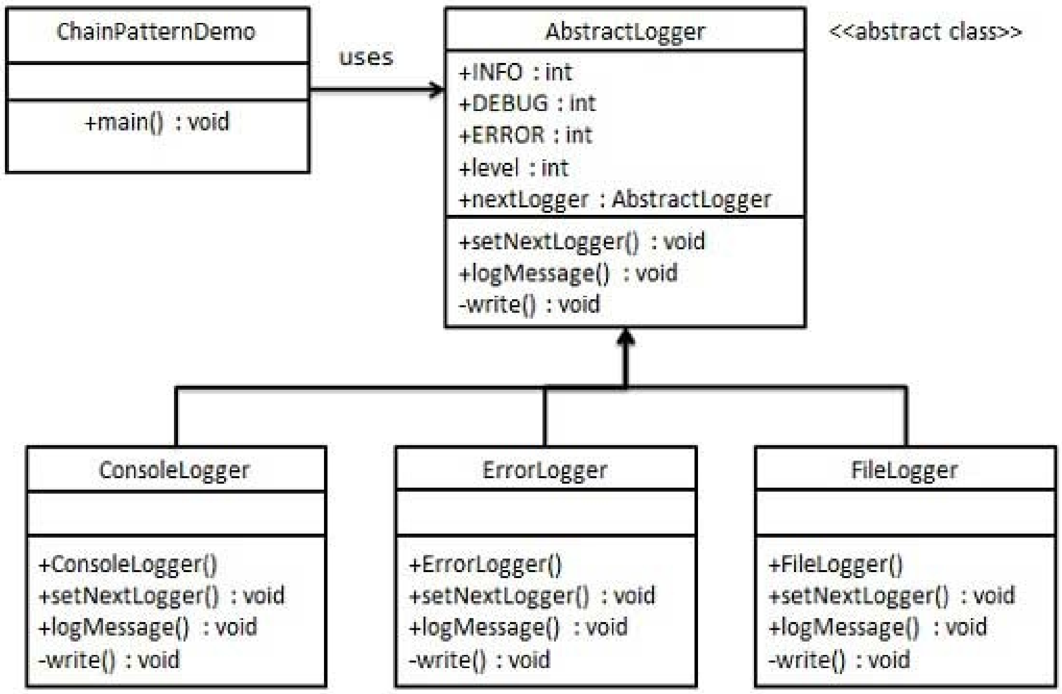
缺点： 1、不能保证请求一定被接收。 2、系统性能将受到一定影响，而且在进行代码调试时不太方便，可能会造成循环调用。 3、可能不容易观察运行时的特征，有碍于除错。

使用场景： 1、有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。 2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。 3、可动态指定一组对象处理请求。

注意事项：在 JAVA WEB 中遇到很多应用。

实现

我们创建抽象类 *AbstractLogger*，带有详细的日志记录级别。然后我们创建三种类型的记录器，都扩展了 *AbstractLogger*。每个记录器消息的级别是否属于自己的级别，如果是则相应地打印出来，否则将不打印并把消息传给下一个记录器。



步骤 1

创建抽象的记录器类。

AbstractLogger.java

```
public abstract class AbstractLogger {
   public static int INFO = 1;
   public static int DEBUG = 2;
   public static int ERROR = 3;

   protected int level;

   //责任链中的下一个元素
   protected AbstractLogger nextLogger;

   public void setNextLogger(AbstractLogger nextLogger){
      this.nextLogger = nextLogger;
   }

   public void logMessage(int level, String message){
      if(this.level <= level){
         write(message);
      }
      if(nextLogger !=null){
         nextLogger.logMessage(level, message);
      }
   }

   abstract protected void write(String message);

}
```

步骤 2

创建扩展了该记录器类的实体类。

ConsoleLogger.java

```
public class ConsoleLogger extends AbstractLogger {

   public ConsoleLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("Standard Console::Logger: " + message);
   }
}
```

ErrorLogger.java

```
public class ErrorLogger extends AbstractLogger {

   public ErrorLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("Error Console::Logger: " + message);
   }
}
```

FileLogger.java

```
public class FileLogger extends AbstractLogger {

   public FileLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("File::Logger: " + message);
   }
}
```

步骤 3

创建不同类型的记录器。赋予它们不同的错误级别，并在每个记录器中设置下一个记录器。每个记录器中的下一个记录器代表的是链的一部分。

ChainPatternDemo.java

```
public class ChainPatternDemo {

   private static AbstractLogger getChainOfLoggers(){

      AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
      AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
      AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);

      errorLogger.setNextLogger(fileLogger);
      fileLogger.setNextLogger(consoleLogger);

      return errorLogger;
   }

   public static void main(String[] args) {
      AbstractLogger loggerChain = getChainOfLoggers();

      loggerChain.logMessage(AbstractLogger.INFO,
         "This is an information.");

      loggerChain.logMessage(AbstractLogger.DEBUG,
         "This is an debug level information.");

      loggerChain.logMessage(AbstractLogger.ERROR,
         "This is an error information.");
   }
}
```

步骤 4

验证输出。

```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error information.
```

命令模式

命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

介绍

意图：将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。

主要解决：在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适。

何时使用：在某些场合，比如要对行为进行"记录、撤销/重做、事务"等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将"行为请求者"与"行为实现者"解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

如何解决：通过调用者调用接受者执行命令，顺序：调用者→接受者→命令。

关键代码：定义三个角色：1、received 真正的命令执行对象 2、Command 3、invoker 使用命令对象的入口

应用实例：struts 1 中的 action 核心控制器 ActionServlet 只有一个，相当于 Invoker，而模型层的类会随着不同的应用有不同的模型类，相当于具体的 Command。

优点： 1、降低了系统耦合度。 2、新的命令可以很容易添加到系统中去。

缺点：使用命令模式可能会导致某些系统有过多的具体命令类。

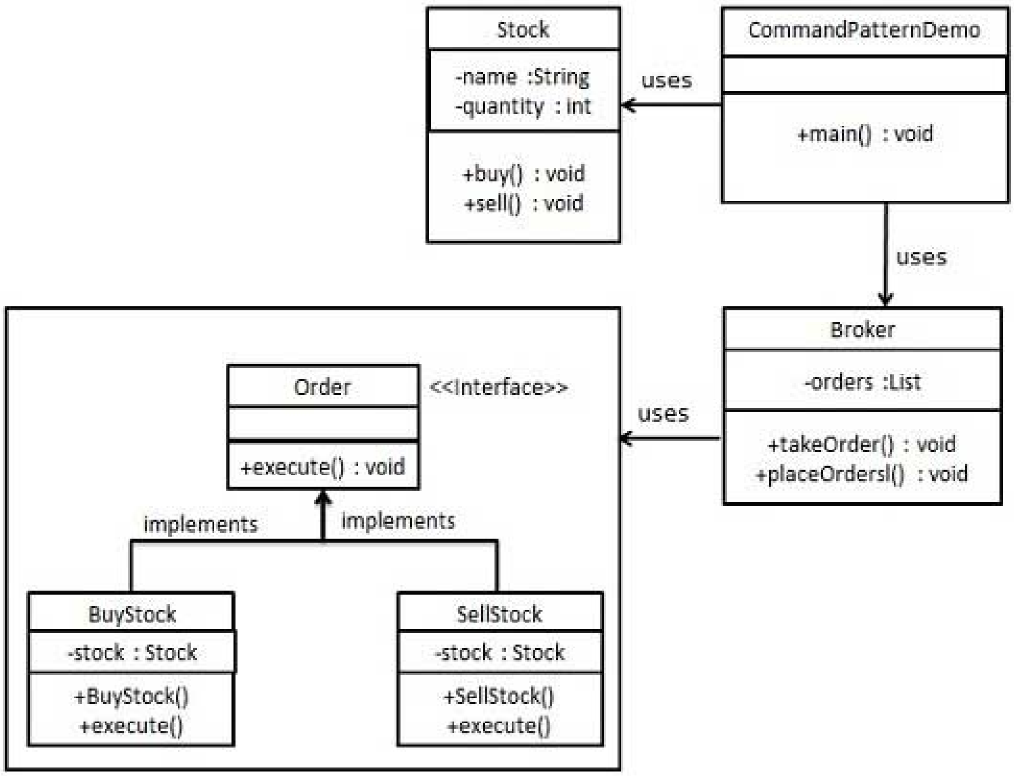
使用场景：认为是命令的地方都可以使用命令模式，比如： 1、GUI 中每一个按钮都是一条命令。 2、模拟 CMD。

注意事项：系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作，也可以考虑使用命令模式，见命令模式的扩展。

实现

我们首先创建作为命令的接口 *Order*，然后创建作为请求的 *Stock* 类。实体命令类 *BuyStock* 和 *SellStock*，实现了 *Order* 接口，将执行实际的命令处理。创建作为调用对象的类 *Broker*，它接受订单并能下订单。

Broker 对象使用命令模式，基于命令的类型确定哪个对象执行哪个命令。*CommandPatternDemo*，我们的演示类使用 *Broker* 类来演示命令模式。



步骤 1

创建一个命令接口。

Order.java

```
public interface Order {
   void execute();
}
```

步骤 2

创建一个请求类。

Stock.java

```
public class Stock {

   private String name = "ABC";
   private int quantity = 10;

   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```

步骤 3

创建实现了 *Order* 接口的实体类。

BuyStock.java

```
public class BuyStock implements Order {
   private Stock abcStock;

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```
public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

步骤 4

创建命令调用类。

Broker.java

```
import java.util.ArrayList;
import java.util.List;

   public class Broker {
   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){
      orderList.add(order);
   }

   public void placeOrders(){
      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```

步骤 5

使用 Broker 类来接受并执行命令。

CommandPatternDemo.java

```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);

      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);

      broker.placeOrders();
   }
}
```

步骤 6

验证输出。

```
Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold
```

解释器模式

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。

介绍

意图：给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。

主要解决：对于一些固定文法构建一个解释句子的解释器。

何时使用：如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

如何解决：构件语法树，定义终结符与非终结符。

关键代码：构件环境类，包含解释器之外的一些全局信息，一般是 HashMap。

应用实例：编译器、运算表达式计算。

优点： 1、可扩展性比较好，灵活。 2、增加了新的解释表达式的方式。 3、易于实现简单文法。

缺点： 1、可利用场景比较少。 2、对于复杂的文法比较难维护。 3、解释器模式会引起类膨胀。 4、解释器模式采用递归调用方法。

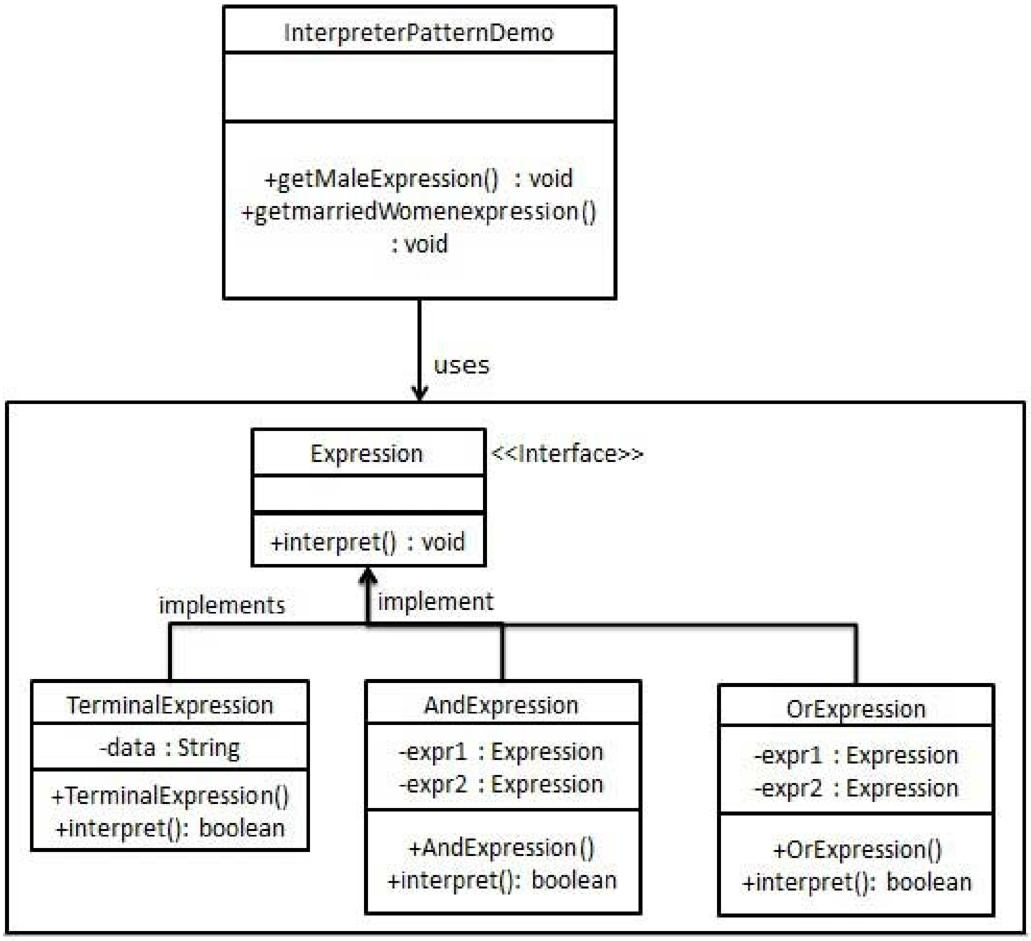
使用场景： 1、可以将一个需要解释执行的语言中的句子表示为一个抽象语法树。 2、一些重复出现的问题可以用一种简单的语言来进行表达。 3、一个简单语法需要解释的场景。

注意事项：可利用场景比较少，JAVA 中如果碰到可以用 expression4J 代替。

实现

我们将创建一个接口 *Expression* 和实现了 *Expression* 接口的实体类。定义作为上下文中主要解释器的 *TerminalExpression* 类。其他的类 *OrExpression*、*AndExpression* 用于创建组合式表达式。

InterpreterPatternDemo，我们的演示类使用 *Expression* 类创建规则和演示表达式的解析。



步骤 1

创建一个表达式接口。

Expression.java

```
public interface Expression {
   public boolean interpret(String context);
}
```

步骤 2

创建实现了上述接口的实体类。

TerminalExpression.java

```
public class TerminalExpression implements Expression {

   private String data;

   public TerminalExpression(String data){
      this.data = data;
   }

   @Override
   public boolean interpret(String context) {
      if(context.contains(data)){
         return true;
      }
      return false;
   }
}
```

OrExpression.java

```
public class OrExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public OrExpression(Expression expr1, Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) || expr2.interpret(context);
   }
}
```

AndExpression.java

```
public class AndExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public AndExpression(Expression expr1, Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) &amp;&amp; expr2.interpret(context);
   }
}
```

步骤 3

InterpreterPatternDemo 使用 *Expression* 类来创建规则，并解析它们。

InterpreterPatternDemo.java

```
public class InterpreterPatternDemo {

   //规则：Robert 和 John 是男性
   public static Expression getMaleExpression(){
      Expression robert = new TerminalExpression("Robert");
      Expression john = new TerminalExpression("John");
      return new OrExpression(robert, john);
   }

   //规则：Julie 是一个已婚的女性
   public static Expression getMarriedWomanExpression(){
      Expression julie = new TerminalExpression("Julie");
      Expression married = new TerminalExpression("Married");
      return new AndExpression(julie, married);
   }

   public static void main(String[] args) {
      Expression isMale = getMaleExpression();
      Expression isMarriedWoman = getMarriedWomanExpression();

      System.out.println("John is male? " + isMale.interpret("John"));
      System.out.println("Julie is a married women? "
      + isMarriedWoman.interpret("Married Julie"));
   }
}
```

步骤 4

验证输出。

```
John is male? true
Julie is a married women? true
```

迭代器模式

迭代器模式（Iterator Pattern）是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。

迭代器模式属于行为型模式。

介绍

意图：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

主要解决：不同的方式来遍历整个整合对象。

何时使用：遍历一个聚合对象。

如何解决：把在元素之间游走的责任交给迭代器，而不是聚合对象。

关键代码：定义接口：hasNext, next。

应用实例：JAVA 中的 iterator。

优点： 1、它支持以不同的方式遍历一个聚合对象。 2、迭代器简化了聚合类。 3、在同一个聚合上可以有多个遍历。 4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

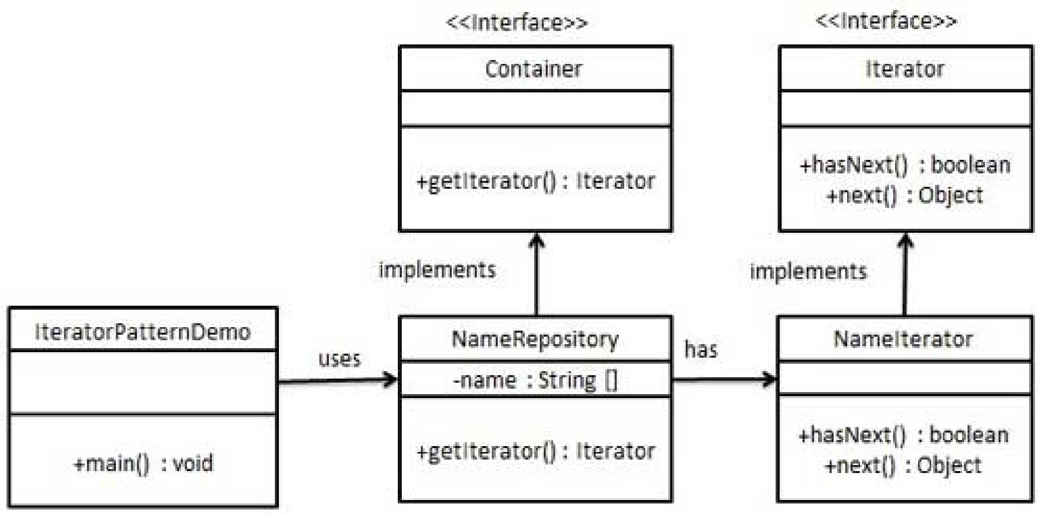
使用场景： 1、访问一个聚合对象的内容而无须暴露它的内部表示。 2、需要为聚合对象提供多种遍历方式。 3、为遍历不同的聚合结构提供一个统一的接口。

注意事项：迭代器模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

实现

我们将创建一个叙述导航方法的 *Iterator* 接口和一个返回迭代器的 *Container* 接口。实现了 *Container* 接口的实体类将负责实现 *Iterator* 接口。

IteratorPatternDemo，我们的演示类使用实体类 *NamesRepository* 来打印 *NamesRepository* 中存储为集合的 *Names*。



步骤 1

创建接口。

Iterator.java

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

步骤 2

创建实现了 *Container* 接口的实体类。该类有实现了 *Iterator* 接口的内部类 *NameIterator*。

NameRepository.java

```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {
         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {
         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

步骤 3

使用 *NameRepository* 来获取迭代器，并打印名字。

IteratorPatternDemo.java

```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

步骤 4

验证输出。

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```

中介者模式

中介者模式（Mediator Pattern）是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。中介者模式属于行为型模式。

介绍

意图：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

主要解决：对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。

何时使用：多个类相互耦合，形成了网状结构。

如何解决：将上述网状结构分离为星型结构。

关键代码：对象 Colleague 之间的通信封装到一个类中单独处理。

应用实例： 1、中国加入 WTO 之前是各个国家相互贸易，结构复杂，现在是各个国家通过 WTO 来互相贸易。 2、机场调度系统。 3、MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者。

优点： 1、降低了类的复杂度，将一对多转化成了一对一。 2、各个类之间的解耦。 3、符合迪米特原则。

缺点：中介者会庞大，变得复杂难以维护。

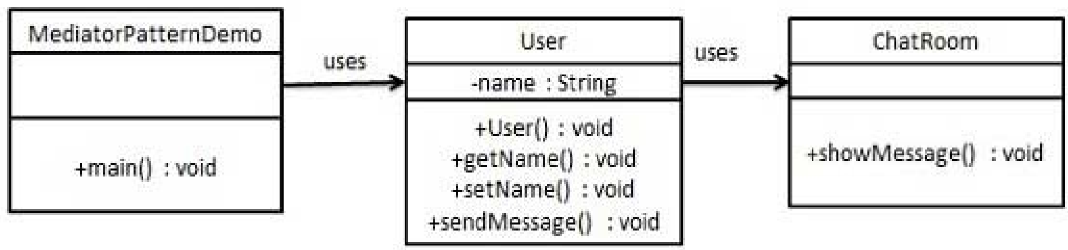
使用场景： 1、系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。 2、想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

注意事项：不应当在职责混乱的时候使用。

实现

我们通过聊天室实例来演示中介者模式。实例中，多个用户可以向聊天室发送消息，聊天室向所有的用户显示消息。我们将创建两个类 *ChatRoom* 和 *User*。*User* 对象使用 *ChatRoom* 方法来分享他们的消息。

MediatorPatternDemo，我们的演示类使用 *User* 对象来显示他们之间的通信。



步骤 1

创建中介类。

ChatRoom.java

```
import java.util.Date;

public class ChatRoom {
   public static void showMessage(User user, String message){
      System.out.println(new Date().toString()
         + " [" + user.getName() +"] : " + message);
   }
}
```

步骤 2

创建 user 类。

User.java

```
public class User {
   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public User(String name){
      this.name  = name;
   }

   public void sendMessage(String message){
      ChatRoom.showMessage(this,message);
   }
}
```

步骤 3

使用 *User* 对象来显示他们之间的通信。

MediatorPatternDemo.java

```
public class MediatorPatternDemo {
   public static void main(String[] args) {
      User robert = new User("Robert");
      User john = new User("John");

      robert.sendMessage("Hi! John!");
      john.sendMessage("Hello! Robert!");
   }
}
```

步骤 4

验证输出。

```
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
```

备忘录模式

备忘录模式（Memento Pattern）保存一个对象的某个状态，以便在适当的时候恢复对象。备忘录模式属于行为型模式。

介绍

意图：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

主要解决：所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。

何时使用：很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有"后悔药"可吃。

如何解决：通过一个备忘录类专门存储对象状态。

关键代码：客户不与备忘录类耦合，与备忘录管理类耦合。

应用实例： 1、后悔药。 2、打游戏时的存档。 3、Windows 里的 ctri + z。 4、IE 中的后退。 4、数据库的事务管理。

优点： 1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。 2、实现了信息的封装，使得用户不需要关心状态的保存细节。

缺点：消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

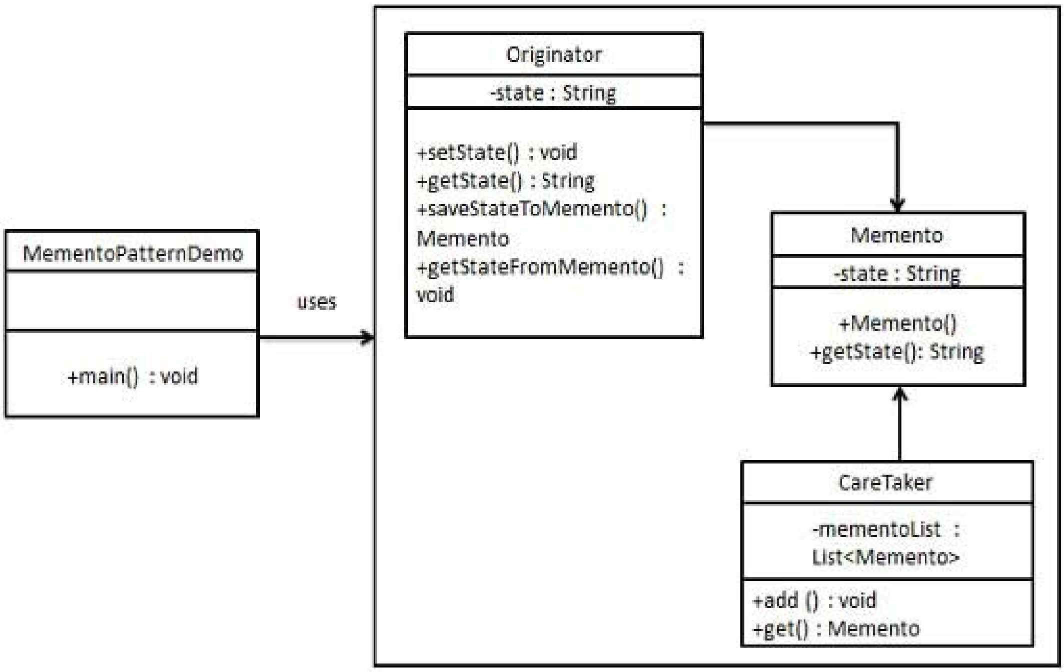
使用场景： 1、需要保存/恢复数据的相关状态场景。 2、提供一个可回滚的操作。

注意事项： 1、为了符合迪米特原则，还要增加一个管理备忘录的类。 2、为了节约内存，可使用原型模式+备忘录模式。

实现

备忘录模式使用三个类 *Memento*、*Originator* 和 *CareTaker*。Memento 包含了要被恢复的对象的状态。Originator 创建并在 Memento 对象中存储状态。Caretaker 对象负责从 Memento 中恢复对象的状态。

MementoPatternDemo，我们的演示类使用 *CareTaker* 和 *Originator* 对象来显示对象的状态恢复。



步骤 1

创建 Memento 类。

Memento.java

```
public class Memento {
   private String state;

   public Memento(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }
}
```

步骤 2

创建 Originator 类。

Originator.java

```
public class Originator {
   private String state;

   public void setState(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }

   public Memento saveStateToMemento(){
      return new Memento(state);
   }

   public void getStateFromMemento(Memento Memento){
      state = Memento.getState();
   }
}
```

步骤 3

创建 CareTaker 类。

CareTaker.java

```
import java.util.ArrayList;
import java.util.List;

public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();

   public void add(Memento state){
      mementoList.add(state);
   }

   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

步骤 4

使用 *CareTaker* 和 *Originator* 对象。

MementoPatternDemo.java

```
public class MementoPatternDemo {
   public static void main(String[] args) {
      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();
      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());
      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());
      originator.setState("State #4");

      System.out.println("Current State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " + originator.getState());
   }
}
```

步骤 5

验证输出。

```
Current State: State #4
First saved State: State #2
Second saved State: State #3
```

观察者模式

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

介绍

意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

如何解决：使用面向对象技术，可以将这种依赖关系弱化。

关键代码：在抽象类里有一个 ArrayList 存放观察者们。

应用实例： 1、拍卖的时候，拍卖师观察最高标价，然后通知给其他竞价者竞价。 2、西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作。

优点： 1、观察者和被观察者是抽象耦合的。 2、建立一套触发机制。

缺点： 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。 2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。 3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

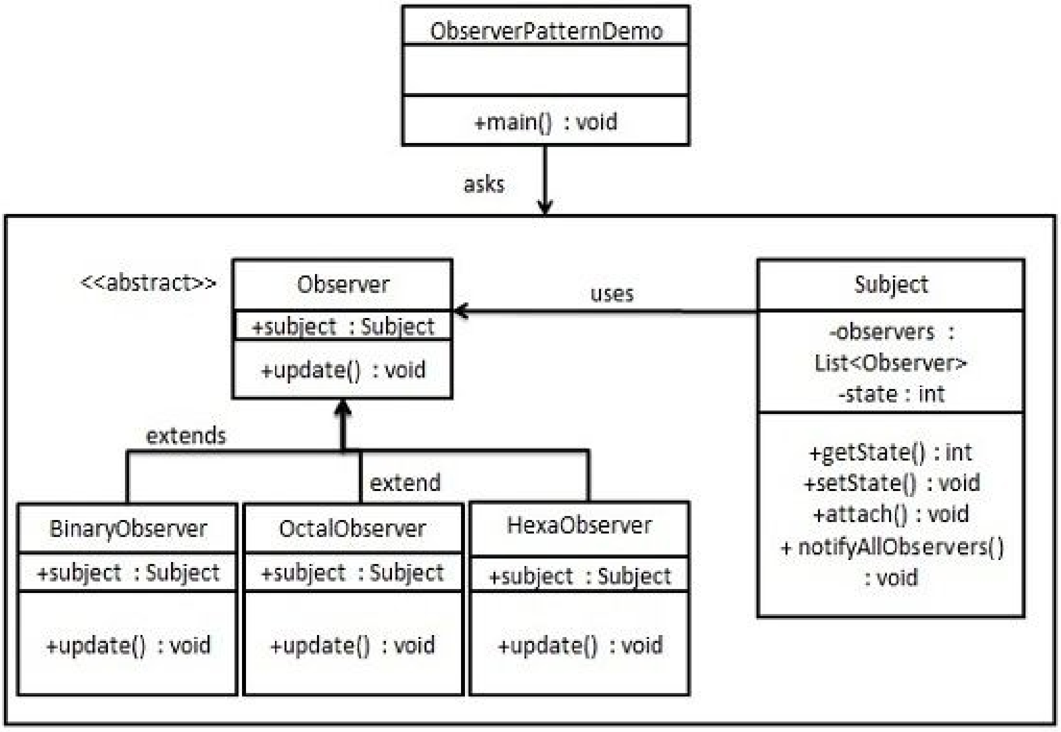
使用场景： 1、有多个子类共有的方法，且逻辑相同。 2、重要的、复杂的方法，可以考虑作为模板方法。

注意事项： 1、JAVA 中已经有了对观察者模式的支持类。 2、避免循环引用。 3、如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。

实现

观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法。我们创建 *Subject* 类、*Observer* 抽象类和扩展了抽象类 *Observer* 的实体类。

ObserverPatternDemo，我们的演示类使用 *Subject* 和实体类对象来演示观察者模式。



步骤 1

创建 Subject 类。

Subject.java

```
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers
      = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```

步骤 2

创建 Observer 类。

Observer.java

```
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

步骤 3

创建实体观察者类。

BinaryObserver.java

```
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: "
      + Integer.toBinaryString( subject.getState() ) );
   }
}
```

OctalObserver.java

```
public class OctalObserver extends Observer{

   public OctalObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
     System.out.println( "Octal String: "
     + Integer.toOctalString( subject.getState() ) );
   }
}
```

HexaObserver.java

```
public class HexaObserver extends Observer{

   public HexaObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Hex String: "
      + Integer.toHexString( subject.getState() ).toUpperCase() );
   }
}
```

步骤 4

使用 *Subject* 和实体观察者对象。

ObserverPatternDemo.java

```
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

步骤 5

验证输出。

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

状态模式

在状态模式（State Pattern）中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。

在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。

介绍

意图：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

主要解决：对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

何时使用：代码中包含大量与对象状态有关的条件语句。

如何解决：将各种具体的状态类抽象出来。

关键代码：通常命令模式的接口中只有一个方法。而状态模式的接口中有一个或者多个方法。而且，状态模式的实现类的方法，一般返回值，或者是改变实例变量的值。也就是说，状态模式一般和对象的状态有关。实现类的方法有不同的功能，覆盖接口中的方法。状态模式和命令模式一样，也可以用于消除 if...else 等条件选择语句。

应用实例： 1、打篮球的时候运动员可以有正常状态、不正常状态和超常状态。 2、曾侯乙编钟中，'钟是抽象接口','钟A'等是具体状态，'曾侯乙编钟'是具体环境（Context）。

优点： 1、封装了转换规则。 2、枚举可能的状态，在枚举状态之前需要确定状态种类。 3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。 4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。 5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

缺点： 1、状态模式的使用必然会增加系统类和对象的个数。 2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。 3、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

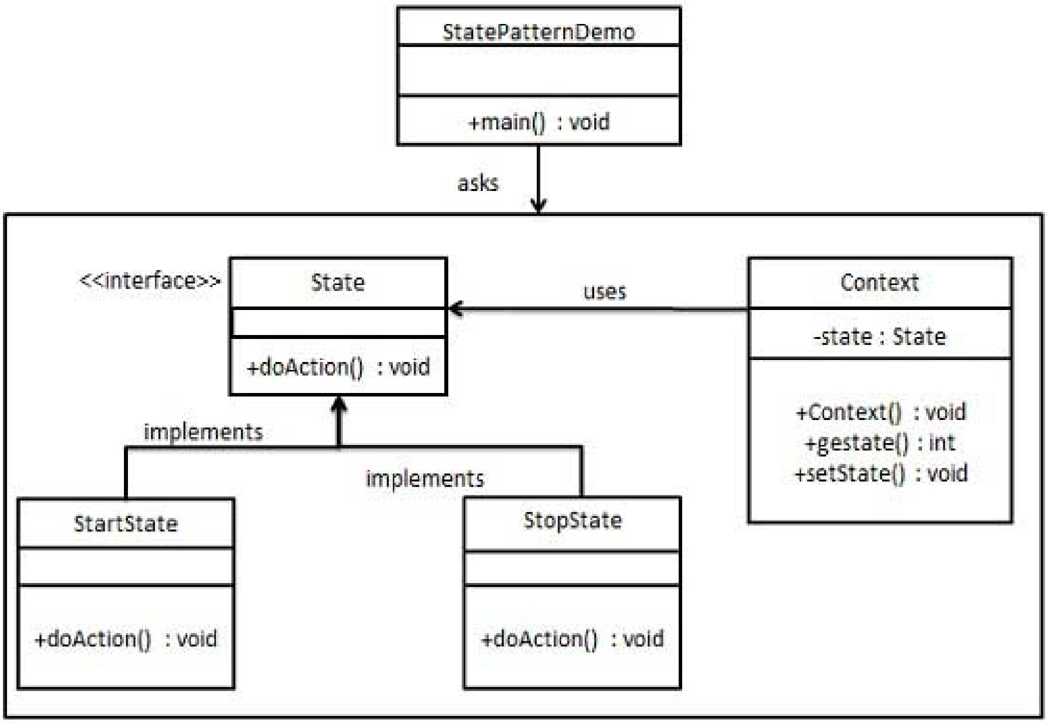
使用场景： 1、行为随状态改变而改变的场景。 2、条件、分支语句的代替者。

注意事项：在行为受状态约束的时候使用状态模式，而且状态不超过 5 个。

实现

我们将创建一个 *State* 接口和实现了 *State* 接口的实体状态类。*Context* 是一个带有某个状态的类。

StaePatternDemo，我们的演示类使用 *Context* 和状态对象来演示 Context 在状态改变时的行为变化。



步骤 1

创建一个接口。

Image.java

```
public interface State {
   public void doAction(Context context);
}
```

步骤 2

创建实现接口的实体类。

StartState.java

```
public class StartState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in start state");
      context.setState(this);
   }

   public String toString(){
      return "Start State";
   }
}
```

StopState.java

```
public class StopState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in stop state");
      context.setState(this);
   }

   public String toString(){
      return "Stop State";
   }
}
```

步骤 3

创建 *Context* 类。

Context.java

```
public class Context {
   private State state;

   public Context(){
      state = null;
   }

   public void setState(State state){
      this.state = state;
   }

   public State getState(){
      return state;
   }
}
```

步骤 4

使用 *Context* 来查看当状态 *State* 改变时的行为变化。

StatePatternDemo.java

```
public class StatePatternDemo {
   public static void main(String[] args) {
      Context context = new Context();

      StartState startState = new StartState();
      startState.doAction(context);

      System.out.println(context.getState().toString());

      StopState stopState = new StopState();
      stopState.doAction(context);

      System.out.println(context.getState().toString());
   }
}
```

步骤 5

验证输出。

```
Player is in start state
Start State
Player is in stop state
Stop State
```

空对象模式

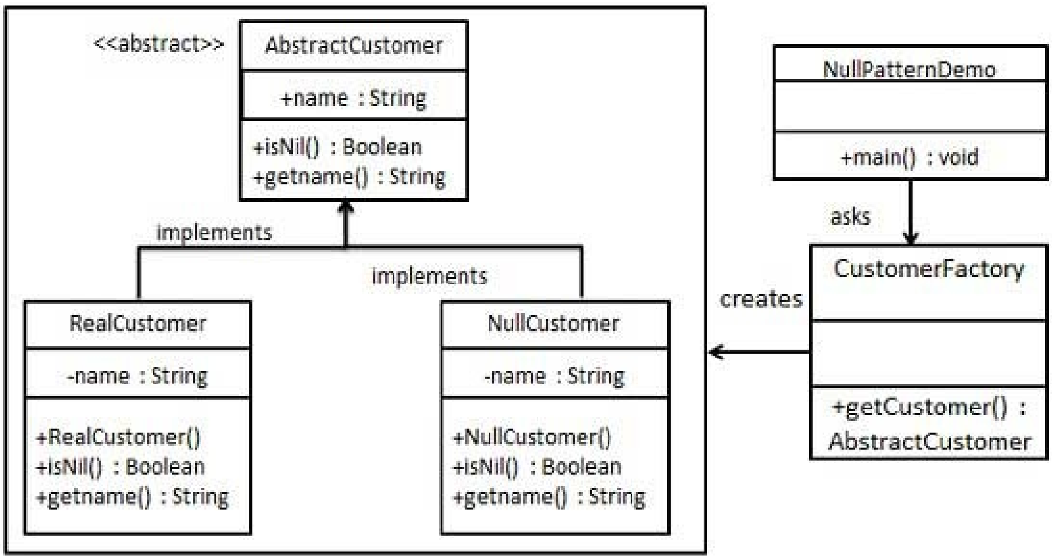
在空对象模式（Null Object Pattern）中，一个空对象取代 NULL 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。

在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

实现

我们将创建一个定义操作（在这里，是客户的名称）的 *AbstractCustomer* 抽象类，和扩展了 *AbstractCustomer* 类的实体类。工厂类 *CustomerFactory* 基于客户传递的名字来返回 *RealCustomer* 或 *NullCustomer* 对象。

NullPatternDemo，我们的演示类使用 *CustomerFactory* 来演示空对象模式的用法。



步骤 1

创建一个抽象类。

AbstractCustomer.java

```
public abstract class AbstractCustomer {
   protected String name;
   public abstract boolean isNil();
   public abstract String getName();
}
```

步骤 2

创建扩展了上述类的实体类。

RealCustomer.java

```
public class RealCustomer extends AbstractCustomer {

   public RealCustomer(String name) {
      this.name = name;
   }

   @Override
   public String getName() {
      return name;
   }

   @Override
   public boolean isNil() {
      return false;
   }
}
```

NullCustomer.java

```
public class NullCustomer extends AbstractCustomer {

   @Override
   public String getName() {
      return "Not Available in Customer Database";
   }

   @Override
   public boolean isNil() {
      return true;
   }
}
```

步骤 3

创建 *CustomerFactory* 类。

CustomerFactory.java

```
public class CustomerFactory {

   public static final String[] names = {"Rob", "Joe", "Julie"};

   public static AbstractCustomer getCustomer(String name){
      for (int i = 0; i < names.length; i++) {
         if (names[i].equalsIgnoreCase(name)){
            return new RealCustomer(name);
         }
      }
      return new NullCustomer();
   }
}
```

步骤 4

使用 *CustomerFactory*，基于客户传递的名字，来获取 *RealCustomer* 或 *NullCustomer* 对象。

NullPatternDemo.java

```
public class NullPatternDemo {
   public static void main(String[] args) {

      AbstractCustomer customer1 = CustomerFactory.getCustomer("Rob");
      AbstractCustomer customer2 = CustomerFactory.getCustomer("Bob");
      AbstractCustomer customer3 = CustomerFactory.getCustomer("Julie");
      AbstractCustomer customer4 = CustomerFactory.getCustomer("Laura");

      System.out.println("Customers");
      System.out.println(customer1.getName());
      System.out.println(customer2.getName());
      System.out.println(customer3.getName());
      System.out.println(customer4.getName());
   }
}
```

步骤 5

验证输出。

```
Customers
Rob
Not Available in Customer Database
Julie
Not Available in Customer Database
```

策略模式

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

介绍

意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

应用实例： 1、诸葛亮的锦囊妙计，每一个锦囊就是一个策略。 2、旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略。 3、JAVA AWT 中的 LayoutManager。

优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。

缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

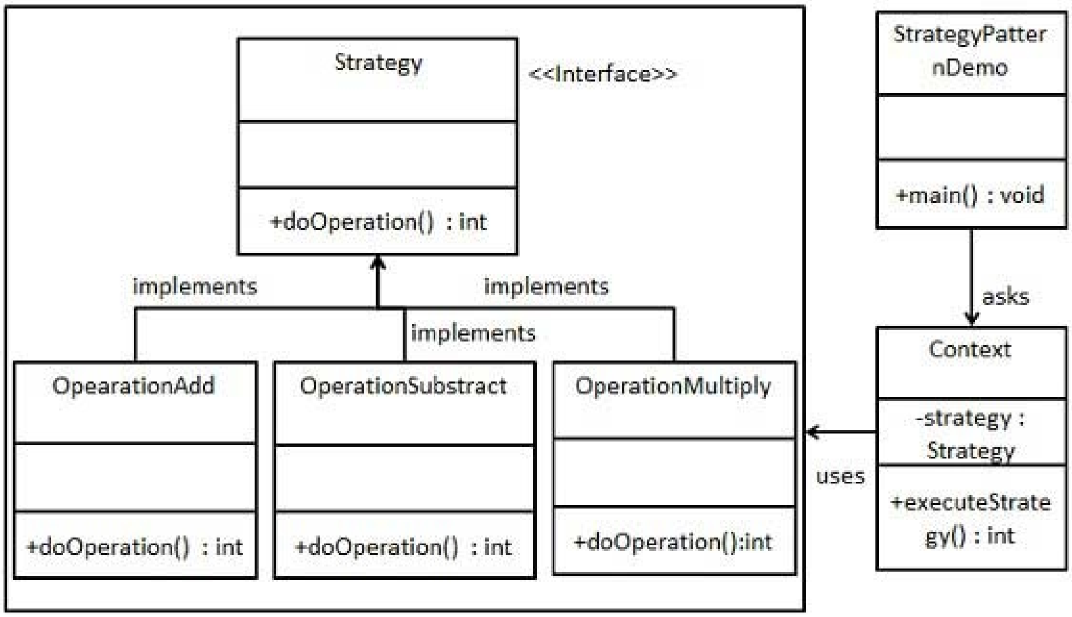
使用场景： 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。 2、一个系统需要动态地在几种算法中选择一种。 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

注意事项：如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

实现

我们将创建一个定义活动的 *Strategy* 接口和实现了 *Strategy* 接口的实体策略类。*Context* 是一个使用了某种策略的类。

StrategyPatternDemo，我们的演示类使用 *Context* 和策略对象来演示 Context 在它所配置或使用的策略改变时的行为变化。



步骤 1

创建一个接口。

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

步骤 2

创建实现接口的实体类。

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

步骤 3

创建 *Context* 类。

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

步骤 4

使用 *Context* 来查看当它改变策略 *Strategy* 时的行为变化。

StatePatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

步骤 5

验证输出。

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

模板模式

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

介绍

意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

主要解决：一些方法通用，却在每一个子类都重新写了这一方法。

何时使用：有一些通用的方法。

如何解决：将这些通用算法抽象出来。

关键代码：在抽象类实现，其他步骤在子类实现。

应用实例： 1、在造房子的时候，地基、走线、水管都一样，只有在建筑的后期才有加壁橱加栅栏等差异。 2、西游记里面菩萨定好的 81 难，这就是一个顶层的逻辑骨架。 3、Spirng 中对 Hibernate 的支持，将一些已经定好的方法封装起来，比如开启事务、获取 Session、关闭 Session 等，程序员不重复写那些已经规范好的代码，直接丢一个实体就可以保存。

优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

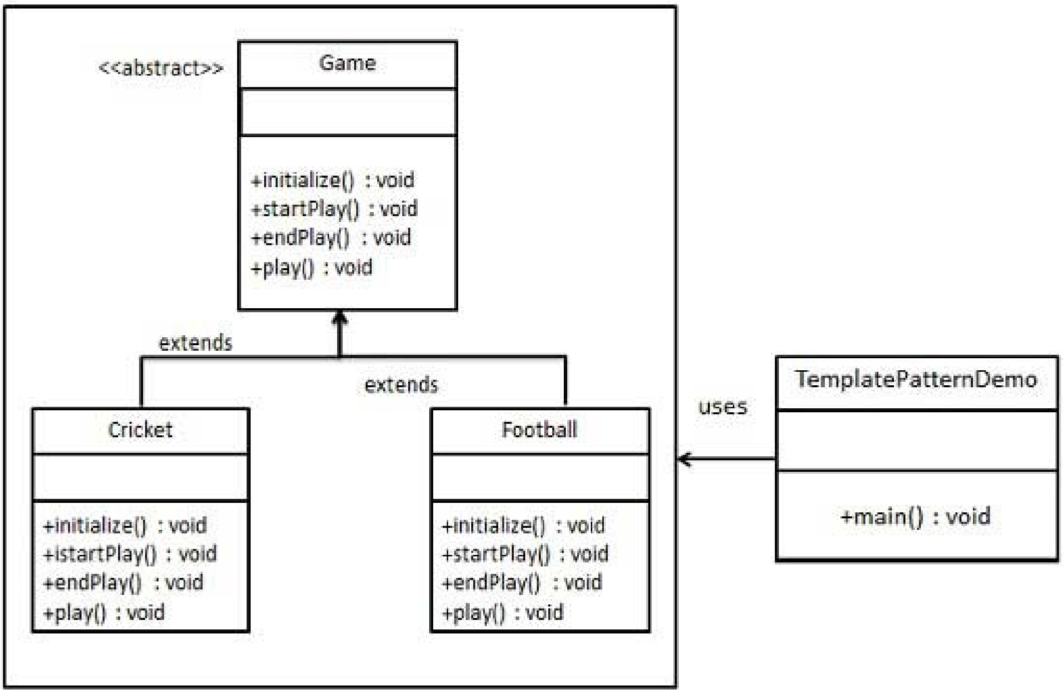
使用场景： 1、有多个子类共有的方法，且逻辑相同。 2、重要的、复杂的方法，可以考虑作为模板方法。

注意事项：为防止恶意操作，一般模板方法都加上 final 关键词。

实现

我们将创建一个定义操作的 *Game* 抽象类，其中，模板方法设置为 final，这样它就不会被重写。*Cricket* 和 *Football* 是扩展了 *Game* 的实体类，它们重写了抽象类的方法。

TemplatePatternDemo，我们的演示类使用 *Game* 来演示模板模式的用法。



步骤 1

创建一个抽象类，它的模板方法被设置为 final。

Game.java

```
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //模板
   public final void play(){

      //初始化游戏
      initialize();

      //开始游戏
      startPlay();

      //结束游戏
      endPlay();
   }
}
```

步骤 2

创建扩展了上述类的实体类。

Cricket.java

```
public class Cricket extends Game {

   @Override
   void endPlay() {
      System.out.println("Cricket Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Cricket Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Cricket Game Started. Enjoy the game!");
   }
}
```

Football.java

```
public class Football extends Game {

   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

步骤 3

使用 *Game* 的模板方法 play() 来演示游戏的定义方式。

TemplatePatternDemo.java

```
public class TemplatePatternDemo {
   public static void main(String[] args) {

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

步骤 4

验证输出。

```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!
```

访问者模式

在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

介绍

意图：主要将数据结构与数据操作分离。

主要解决：稳定的数据结构和易变的操作耦合问题。

何时使用：需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。

如何解决：在被访问的类里面加一个对外提供接待访问者的接口。

关键代码：在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。

应用实例：您在朋友家做客，您是访问者，朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式。

优点： 1、符合单一职责原则。 2、优秀的扩展性。 3、灵活性。

缺点： 1、具体元素对访问者公布细节，违反了迪米特原则。 2、具体元素变更比较困难。 3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象。

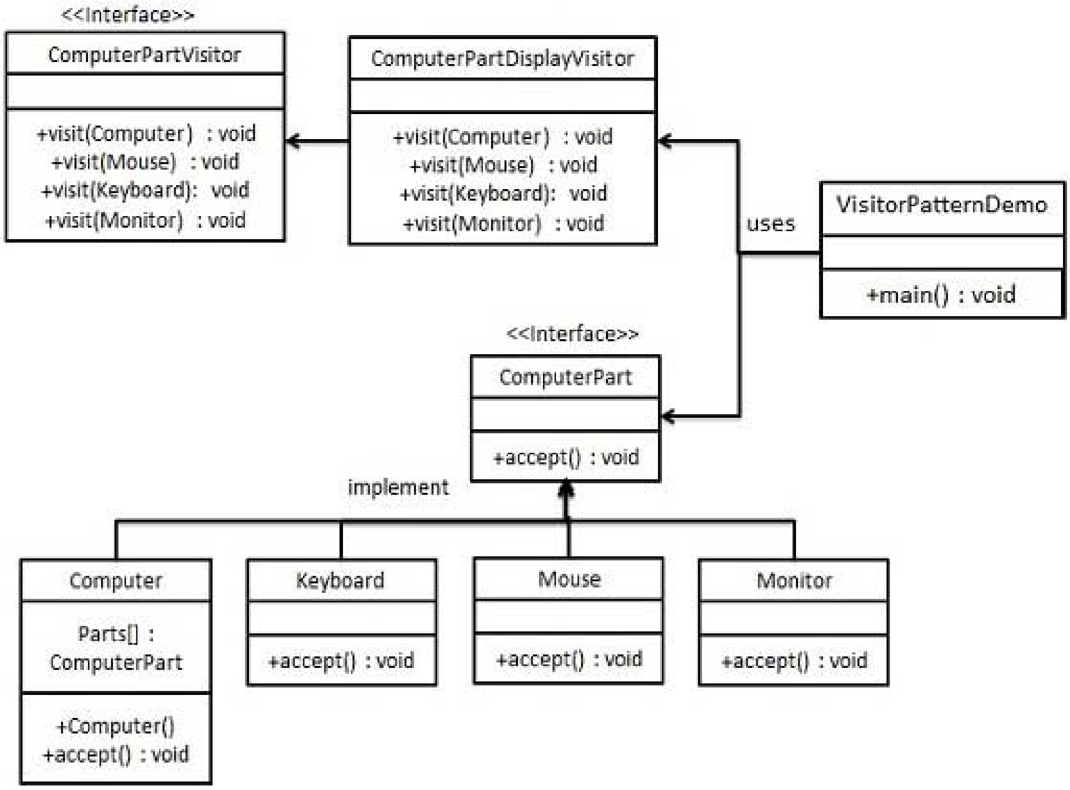
使用场景： 1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

注意事项：访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

实现

我们将创建一个定义接受操作的 *ComputerPart* 接口。*Keyboard*、*Mouse*、*Monitor* 和 *Computer* 是实现了 *ComputerPart* 接口的实体类。我们将定义另一个接口 *ComputerPartVisitor*，它定义了访问者类的操作。*Computer* 使用实体访问者来执行相应的动作。

VisitorPatternDemo，我们的演示类使用 *Computer*、*ComputerPartVisitor* 类来演示访问者模式的用法。



步骤 1

定义一个表示元素的接口。

ComputerPart.java

```
public interface class ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor);
}
```

步骤 2

创建扩展了上述类的实体类。

Keyboard.java

```
public class Keyboard  implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Monitor.java

```
public class Monitor  implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Mouse.java

```
public class Mouse  implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Computer.java

```
public class Computer implements ComputerPart {

   ComputerPart[] parts;

   public Computer(){
      parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
   }

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      for (int i = 0; i < parts.length; i++) {
         parts[i].accept(computerPartVisitor);
      }
      computerPartVisitor.visit(this);
   }
}
```

步骤 3

定义一个表示访问者的接口。

ComputerPartVisitor.java

```
public interface ComputerPartVisitor {
    public void visit(Computer computer);
    public void visit(Mouse mouse);
    public void visit(Keyboard keyboard);
    public void visit(Monitor monitor);
}
```

步骤 4

创建实现了上述类的实体访问者。

ComputerPartDisplayVisitor.java

```
public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

   @Override
   public void visit(Computer computer) {
      System.out.println("Displaying Computer.");
   }

   @Override
   public void visit(Mouse mouse) {
      System.out.println("Displaying Mouse.");
   }

   @Override
   public void visit(Keyboard keyboard) {
      System.out.println("Displaying Keyboard.");
   }

   @Override
   public void visit(Monitor monitor) {
      System.out.println("Displaying Monitor.");
   }
}
```

步骤 5

使用 *ComputerPartDisplayVisitor* 来显示 *Computer* 的组成部分。

VisitorPatternDemo.java

```
public class VisitorPatternDemo {
   public static void main(String[] args) {

      ComputerPart computer = new Computer();
      computer.accept(new ComputerPartDisplayVisitor());
   }
}
```

步骤 6

验证输出。

```
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```

MVC 模式

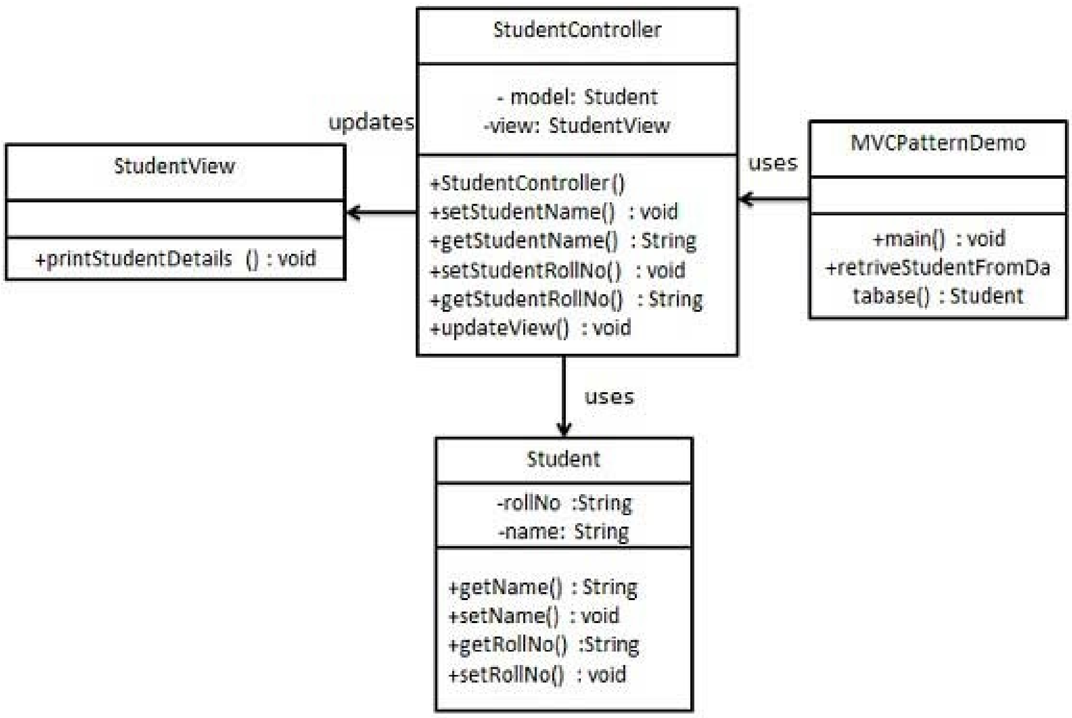
MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

- **Model**（模型） - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
- **View**（视图） - 视图代表模型包含的数据的可视化。
- **Controller**（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

实现

我们将创建一个作为模型的 *Student* 对象。*StudentView* 是一个把学生详细信息输出到控制台的视图类，*StudentController* 是负责存储数据到 *Student* 对象中的控制器类，并相应地更新视图 *StudentView*。

MVCPatternDemo，我们的演示类使用 *StudentController* 来演示 MVC 模式的用法。



步骤 1

创建模型。

Student.java

```
public class Student {
   private String rollNo;
   private String name;
   public String getRollNo() {
      return rollNo;
   }
   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

步骤 2

创建视图。

StudentView.java

```
public class StudentView {
   public void printStudentDetails(String studentName, String studentRollNo){
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " + studentRollNo);
   }
}
```

步骤 3

创建控制器。

StudentController.java

```
public class StudentController {
   private Student model;
   private StudentView view;

   public StudentController(Student model, StudentView view){
      this.model = model;
      this.view = view;
   }

   public void setStudentName(String name){
      model.setName(name);
   }

   public String getStudentName(){
      return model.getName();
   }

   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }

   public String getStudentRollNo(){
      return model.getRollNo();
   }

   public void updateView(){
      view.printStudentDetails(model.getName(), model.getRollNo());
   }
}
```

步骤 4

使用 *StudentController* 方法来演示 MVC 设计模式的用法。

MVCPatternDemo.java

```
public class MVCPatternDemo {
   public static void main(String[] args) {

      //从数据可获取学生记录
      Student model  = retriveStudentFromDatabase();

      //创建一个视图：把学生详细信息输出到控制台
      StudentView view = new StudentView();

      StudentController controller = new StudentController(model, view);

      controller.updateView();

      //更新模型数据
      controller.setStudentName("John");

      controller.updateView();
   }

   private static Student retriveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Robert");
      student.setRollNo("10");
      return student;
   }
}
```

步骤 5

验证输出。

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

业务代表模式

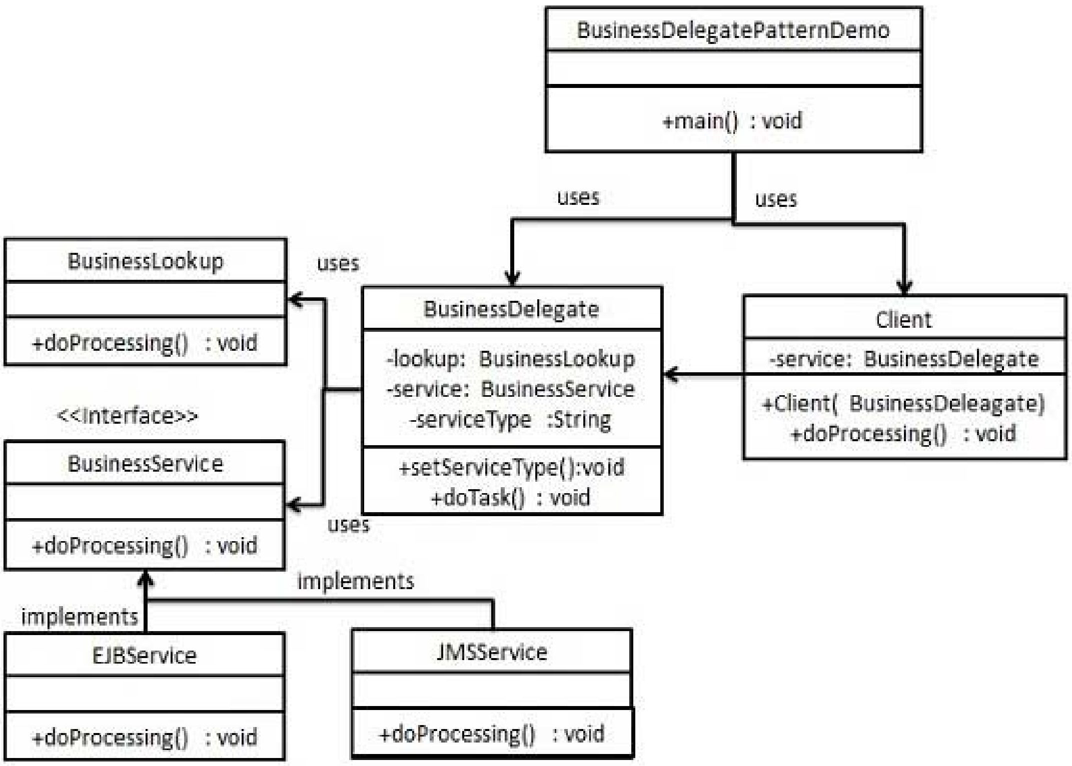
业务代表模式（Business Delegate Pattern）用于对表示层和业务层解耦。它基本上是用来减少通信或对表示层代码中的业务层代码的远程查询功能。在业务层中我们有以下实体。

- 客户端（**Client**） - 表示层代码可以是 JSP、servlet 或 UI java 代码。
- 业务代表（**Business Delegate**） - 一个为客户端实体提供的入口类，它提供了对业务服务方法的访问。
- 查询服务（**LookUp Service**） - 查找服务对象负责获取相关的业务实现，并提供业务对象对业务代表对象的访问。
- 业务服务（**Business Service**） - 业务服务接口。实现了该业务服务的实体类，提供了实际的业务实现逻辑。

实现

我们将创建 *Client*、*BusinessDelegate*、*BusinessService*、*LookUpService*、*JMSService* 和 *EJBService* 来表示业务代表模式中的各种实体。

BusinessDelegatePatternDemo，我们的演示类使用 *BusinessDelegate* 和 *Client* 来演示业务代表模式的用法。



步骤 1

创建 BusinessService 接口。

BusinessService.java

```
public interface BusinessService {
   public void doProcessing();
}
```

步骤 2

创建实体服务类。

EJBService.java

```
public class EJBService implements BusinessService {

   @Override
   public void doProcessing() {
      System.out.println("Processing task by invoking EJB Service");
   }
}
```

JMSService.java

```
public class JMSService implements BusinessService {

   @Override
   public void doProcessing() {
      System.out.println("Processing task by invoking JMS Service");
   }
}
```

步骤 3

创建业务查询服务。

BusinessLookUp.java

```
public class BusinessLookUp {
   public BusinessService getBusinessService(String serviceType){
      if(serviceType.equalsIgnoreCase("EJB")){
         return new EJBService();
      }else {
         return new JMSService();
      }
   }
}
```

步骤 4

创建业务代表。

BusinessDelegate.java

```
public class BusinessDelegate {
   private BusinessLookUp lookupService = new BusinessLookUp();
   private BusinessService businessService;
   private String serviceType;

   public void setServiceType(String serviceType){
      this.serviceType = serviceType;
   }

   public void doTask(){
      businessService = lookupService.getBusinessService(serviceType);
      businessService.doProcessing();
   }
}
```

步骤 5

创建客户端。

Student.java

```
public class Client {

   BusinessDelegate businessService;

   public Client(BusinessDelegate businessService){
      this.businessService  = businessService;
   }

   public void doTask(){
      businessService.doTask();
   }
}
```

步骤 6

使用 BusinessDelegate 和 Client 类来演示业务代表模式。

BusinessDelegatePatternDemo.java

```
public class BusinessDelegatePatternDemo {

   public static void main(String[] args) {

      BusinessDelegate businessDelegate = new BusinessDelegate();
      businessDelegate.setServiceType("EJB");

      Client client = new Client(businessDelegate);
      client.doTask();

      businessDelegate.setServiceType("JMS");
      client.doTask();
   }
}
```

步骤 7

验证输出。

```
Processing task by invoking EJB Service
Processing task by invoking JMS Service
```

组合实体模式

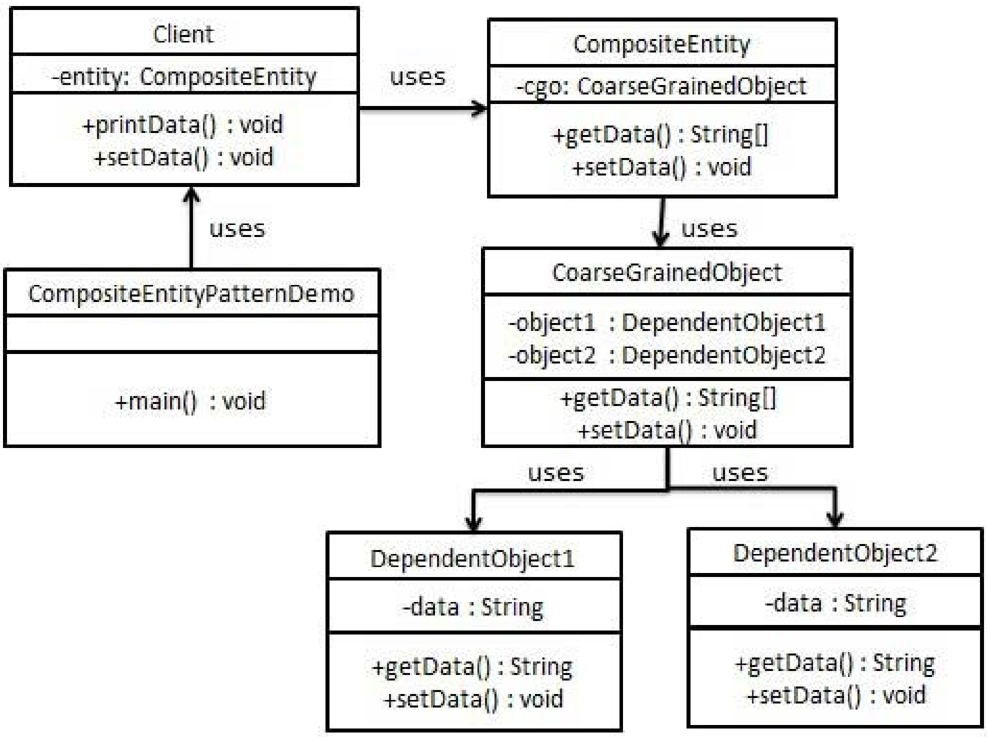
组合实体模式（Composite Entity Pattern）用在 EJB 持久化机制中。一个组合实体是一个 EJB 实体 bean，代表了对象的图解。当更新一个组合实体时，内部依赖对象 beans 会自动更新，因为它们是由 EJB 实体 bean 管理的。以下是组合实体 bean 的参与者。

- 组合实体（**Composite Entity**） - 它是主要的实体 bean。它可以是粗粒的，或者可以包含一个粗粒度对象，用于持续生命周期。
- 粗粒度对象（**Coarse-Grained Object**） - 该对象包含以来对象。它有自己的生命周期，也能管理依赖对象的生命周期。
- 依赖对象（**Dependent Object**） - 依赖对象是一个持续生命周期依赖于粗粒度对象的对象。
- 策略（**Strategies**） - 策略表示如何实现组合实体。

实现

我们将创建作为组合实体的 *CompositeEntity* 对象。*CoarseGrainedObject* 是一个包含依赖对象的类。

CompositeEntityPatternDemo，我们的演示类使用 *Client* 类来演示组合实体模式的用法。



步骤 1

创建依赖对象。

DependentObject1.java

```
public class DependentObject1 {

   private String data;

   public void setData(String data){
      this.data = data;
   }

   public String getData(){
      return data;
   }
}
```

DependentObject2.java

```
public class DependentObject2 {

   private String data;

   public void setData(String data){
      this.data = data;
   }

   public String getData(){
      return data;
   }
}
```

步骤 2

创建粗粒度对象。

CoarseGrainedObject.java

```
public class CoarseGrainedObject {
   DependentObject1 do1 = new DependentObject1();
   DependentObject2 do2 = new DependentObject2();

   public void setData(String data1, String data2){
      do1.setData(data1);
      do2.setData(data2);
   }

   public String[] getData(){
      return new String[] {do1.getData(),do2.getData()};
   }
}
```

步骤 3

创建组合实体。

CompositeEntity.java

```
public class CompositeEntity {
   private CoarseGrainedObject cgo = new CoarseGrainedObject();

   public void setData(String data1, String data2){
      cgo.setData(data1, data2);
   }

   public String[] getData(){
      return cgo.getData();
   }
}
```

步骤 4

创建使用组合实体的客户端类。

Client.java

```
public class Client {
   private CompositeEntity compositeEntity = new CompositeEntity();

   public void printData(){
      for (int i = 0; i < compositeEntity.getData().length; i++) {
         System.out.println("Data: " + compositeEntity.getData()[i]);
      }
   }

   public void setData(String data1, String data2){
      compositeEntity.setData(data1, data2);
   }
}
```

步骤 5

使用 *Client* 来演示组合实体设计模式的用法。

CompositeEntityPatternDemo.java

```
public class CompositeEntityPatternDemo {
   public static void main(String[] args) {
       Client client = new Client();
       client.setData("Test", "Data");
       client.printData();
       client.setData("Second Test", "Data1");
       client.printData();
   }
}
```

步骤 6

验证输出。

```
Data: Test
Data: Data
Data: Second Test
Data: Data1
```

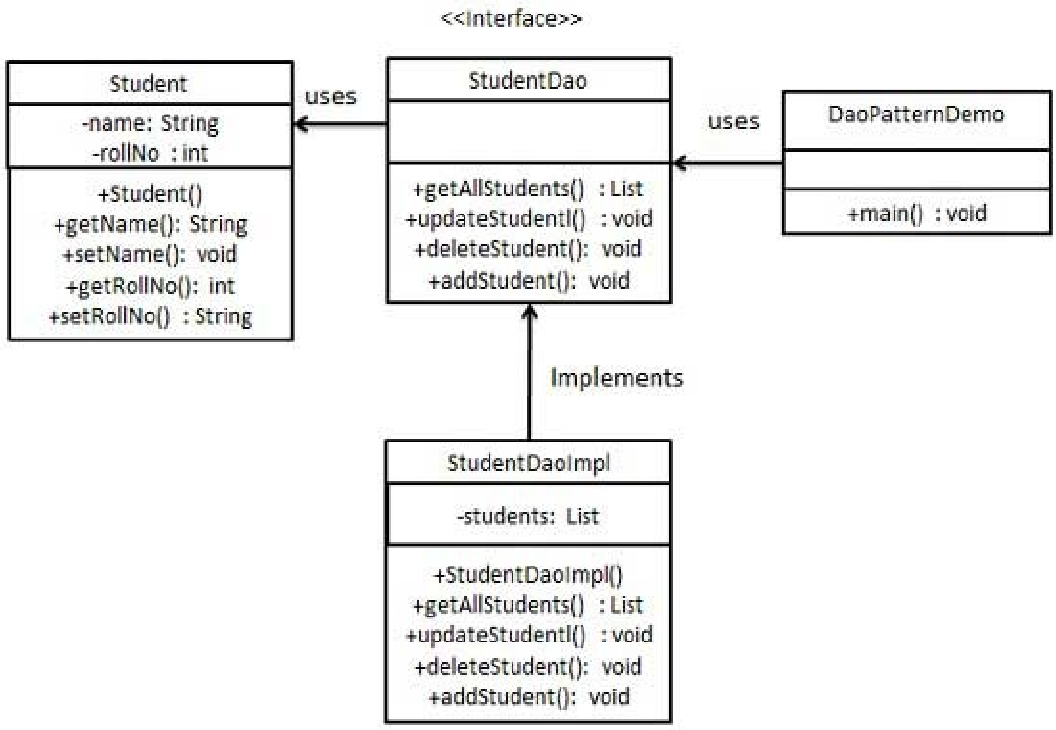
数据访问对象模式

数据访问对象模式（Data Access Object Pattern）或 DAO 模式用于把低级的数据访问 API 或操作从高级的业务服务中分离出来。以下是数据访问对象模式的参与者。

- 数据访问对象接口（**Data Access Object Interface**） - 该接口定义了在一个模型对象上要执行的标准操作。
- 数据访问对象实体类（**Data Access Object concrete class**） - 该类实现了上述的接口。该类负责从数据源获取数据，数据源可以是数据库，也可以是 xml，或者是其他的存储机制。
- 模型对象/数值对象（**Model Object/Value Object**） - 该对象是简单的 POJO，包含了 get/set 方法来存储通过使用 DAO 类检索到的数据。

实现

我们将创建一个作为模型对象或数值对象的 *Student* 对象。*StudentDao* 是数据访问对象接口。*StudentDaoImpl* 是实现了数据访问对象接口的实体类。*DaoPatternDemo*，我们的演示类使用 *StudentDao* 来演示数据访问对象模式的用法。



步骤 1

创建数值对象。

Student.java

```
public class Student {
   private String name;
   private int rollNo;

   Student(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getRollNo() {
      return rollNo;
   }

   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```

步骤 2

创建数据访问对象接口。

StudentDao.java

```
import java.util.List;

public interface StudentDao {
   public List<Student> getAllStudents();
   public Student getStudent(int rollNo);
   public void updateStudent(Student student);
   public void deleteStudent(Student student);
}
```

步骤 3

创建实现了上述接口的实体类。

StudentDaoImpl.java

```
import java.util.ArrayList;
import java.util.List;

public class StudentDaoImpl implements StudentDao {

   //列表是当作一个数据库
   List<Student> students;

   public StudentDaoImpl(){
      students = new ArrayList<Student>();
      Student student1 = new Student("Robert",0);
      Student student2 = new Student("John",1);
      students.add(student1);
      students.add(student2);
   }
   @Override
   public void deleteStudent(Student student) {
      students.remove(student.getRollNo());
      System.out.println("Student: Roll No " + student.getRollNo()
         +", deleted from database");
   }

   //从数据库中检索学生名单
   @Override
   public List<Student> getAllStudents() {
      return students;
   }

   @Override
   public Student getStudent(int rollNo) {
      return students.get(rollNo);
   }

   @Override
   public void updateStudent(Student student) {
      students.get(student.getRollNo()).setName(student.getName());
      System.out.println("Student: Roll No " + student.getRollNo()
         +", updated in the database");
   }
}
```

步骤 4

使用 *StudentDao* 来演示数据访问对象模式的用法。

CompositeEntityPatternDemo.java

```
public class DaoPatternDemo {
   public static void main(String[] args) {
      StudentDao studentDao = new StudentDaoImpl();

      //输出所有的学生
      for (Student student : studentDao.getAllStudents()) {
         System.out.println("Student: [RollNo : "
            +student.getRollNo()+", Name : "+student.getName()+" ]");
      }

      //更新学生
      Student student =studentDao.getAllStudents().get(0);
      student.setName("Michael");
      studentDao.updateStudent(student);

      //获取学生
      studentDao.getStudent(0);
      System.out.println("Student: [RollNo : "
         +student.getRollNo()+", Name : "+student.getName()+" ]");
   }
}
```

步骤 5

验证输出。

```
Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]
```

前端控制器模式

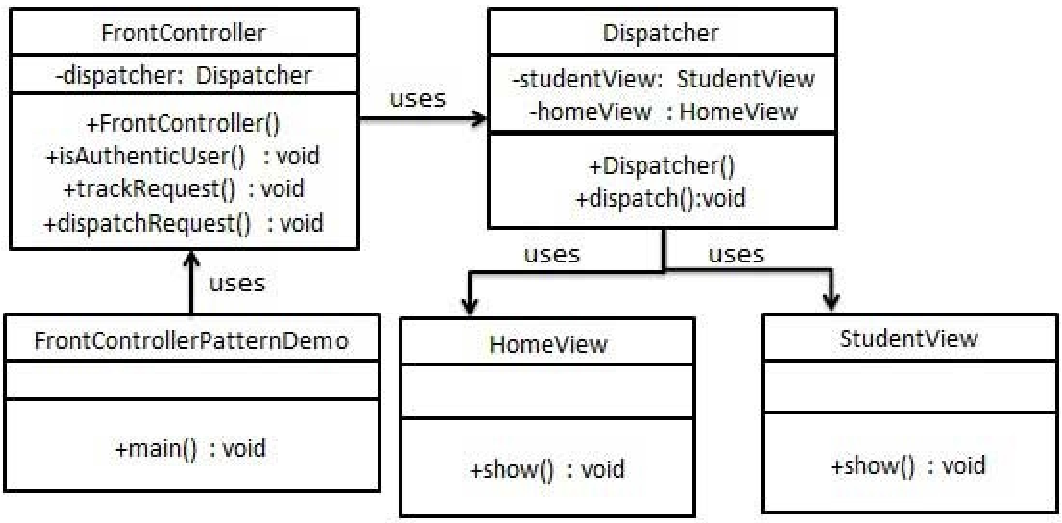
前端控制器模式（Front Controller Pattern）是用来提供一个集中的请求处理机制，所有的请求都将由一个单一的处理程序处理。该处理程序可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

- 前端控制器（**Front Controller**） - 处理应用程序所有类型请求的单个处理程序，应用程序可以是基于 web 的应用程序，也可以是基于桌面的应用程序。
- 调度器（**Dispatcher**） - 前端控制器可能使用一个调度器对象来调度请求到相应的具体处理程序。
- 视图（**View**） - 视图是为请求而创建的对象。

实现

我们将创建 *FrontController*、*Dispatcher* 分别当作前端控制器和调度器。*HomeView* 和 *StudentView* 表示各种为前端控制器接收到的请求而创建的视图。

FrontControllerPatternDemo，我们的演示类使用 *FrontController* 来演示前端控制器设计模式。



步骤 1

创建视图。

HomeView.java

```
public class HomeView {
   public void show(){
      System.out.println("Displaying Home Page");
   }
}
```

StudentView.java

```
public class StudentView {
   public void show(){
      System.out.println("Displaying Student Page");
   }
}
```

步骤 2

创建调度器 Dispatcher。

Dispatcher.java

```
public class Dispatcher {
   private StudentView studentView;
   private HomeView homeView;
   public Dispatcher(){
      studentView = new StudentView();
      homeView = new HomeView();
   }

   public void dispatch(String request){
      if(request.equalsIgnoreCase("STUDENT")){
         studentView.show();
      }else{
         homeView.show();
      }
   }
}
```

步骤 3

创建前端控制器 FrontController。

Context.java

```
public class FrontController {

   private Dispatcher dispatcher;

   public FrontController(){
      dispatcher = new Dispatcher();
   }

   private boolean isAuthenticUser(){
      System.out.println("User is authenticated successfully.");
      return true;
   }

   private void trackRequest(String request){
      System.out.println("Page requested: " + request);
   }

   public void dispatchRequest(String request){
      //记录每一个请求
      trackRequest(request);
      //对用户进行身份验证
      if(isAuthenticUser()){
         dispatcher.dispatch(request);
      }
   }
}
```

步骤 4

使用 *FrontController* 来演示前端控制器设计模式。

FrontControllerPatternDemo.java

```
public class FrontControllerPatternDemo {
   public static void main(String[] args) {
      FrontController frontController = new FrontController();
      frontController.dispatchRequest("HOME");
      frontController.dispatchRequest("STUDENT");
   }
}
```

步骤 5

验证输出。

```
Page requested: HOME
User is authenticated successfully.
Displaying Home Page
Page requested: STUDENT
User is authenticated successfully.
Displaying Student Page
```

拦截过滤器模式

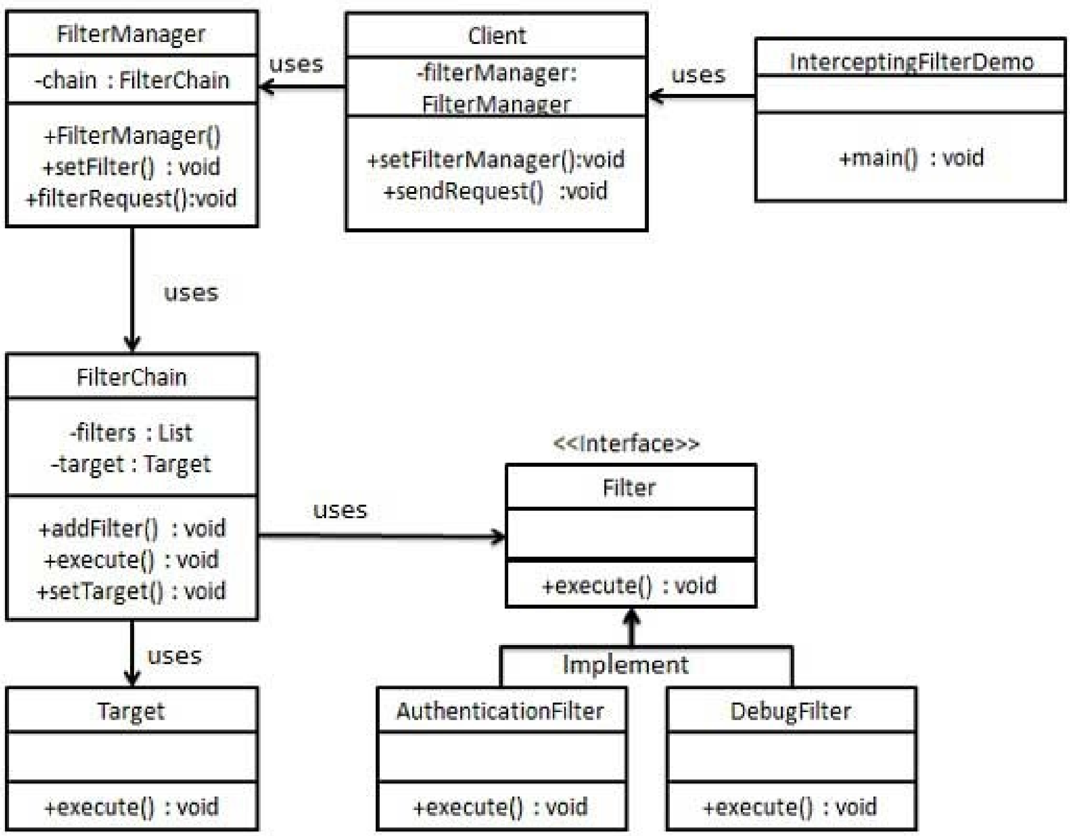
拦截过滤器模式（Intercepting Filter Pattern）用于对应用程序的请求或响应做一些预处理/后处理。定义过滤器，并在把请求传给实际目标应用程序之前应用在请求上。过滤器可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

- 过滤器（**Filter**） - 过滤器在请求处理程序执行请求之前或之后，执行某些任务。
- 过滤器链（**Filter Chain**） - 过滤器链带有多个过滤器，并在 Target 上按照定义的顺序执行这些过滤器。
- **Target** - Target 对象是请求处理程序。
- 过滤管理器（**Filter Manager**） - 过滤管理器管理过滤器和过滤器链。
- 客户端（**Client**） - Client 是向 Target 对象发送请求的对象。

实现

我们将创建 *FilterChain*、*FilterManager*、*Target*、*Client* 作为表示实体的各种对象。*AuthenticationFilter* 和 *DebugFilter* 表示实体过滤器。

InterceptingFilterDemo，我们的演示类使用 *Client* 来演示拦截过滤器设计模式。



步骤 1

创建过滤器接口 Filter。

Filter.java

```
public interface Filter {
   public void execute(String request);
}
```

步骤 2

创建实体过滤器。

AuthenticationFilter.java

```
public class AuthenticationFilter implements Filter {
   public void execute(String request){
      System.out.println("Authenticating request: " + request);
   }
}
```

DebugFilter.java

```
public class DebugFilter implements Filter {
   public void execute(String request){
      System.out.println("request log: " + request);
   }
}
```

步骤 3

创建 Target。

Target.java

```
public class Target {
   public void execute(String request){
      System.out.println("Executing request: " + request);
   }
}
```

步骤 4

创建过滤器链。

FilterChain.java

```
import java.util.ArrayList;
import java.util.List;

public class FilterChain {
   private List<Filter> filters = new ArrayList<Filter>();
   private Target target;

   public void addFilter(Filter filter){
      filters.add(filter);
   }

   public void execute(String request){
      for (Filter filter : filters) {
         filter.execute(request);
      }
      target.execute(request);
   }

   public void setTarget(Target target){
      this.target = target;
   }
}
```

步骤 5

创建过滤管理器。

FilterManager.java

```
public class FilterManager {
   FilterChain filterChain;

   public FilterManager(Target target){
      filterChain = new FilterChain();
      filterChain.setTarget(target);
   }
   public void setFilter(Filter filter){
      filterChain.addFilter(filter);
   }

   public void filterRequest(String request){
      filterChain.execute(request);
   }
}
```

步骤 6

创建客户端 Client。

Client.java

```
public class Client {
   FilterManager filterManager;

   public void setFilterManager(FilterManager filterManager){
      this.filterManager = filterManager;
   }

   public void sendRequest(String request){
      filterManager.filterRequest(request);
   }
}
```

步骤 7

使用 *Client* 来演示拦截过滤器设计模式。

FrontControllerPatternDemo.java

```
public class InterceptingFilterDemo {
   public static void main(String[] args) {
      FilterManager filterManager = new FilterManager(new Target());
      filterManager.setFilter(new AuthenticationFilter());
      filterManager.setFilter(new DebugFilter());

      Client client = new Client();
      client.setFilterManager(filterManager);
      client.sendRequest("HOME");
   }
}
```

步骤 8

验证输出。

```
Authenticating request: HOME
request log: HOME
Executing request: HOME
```

服务定位器模式

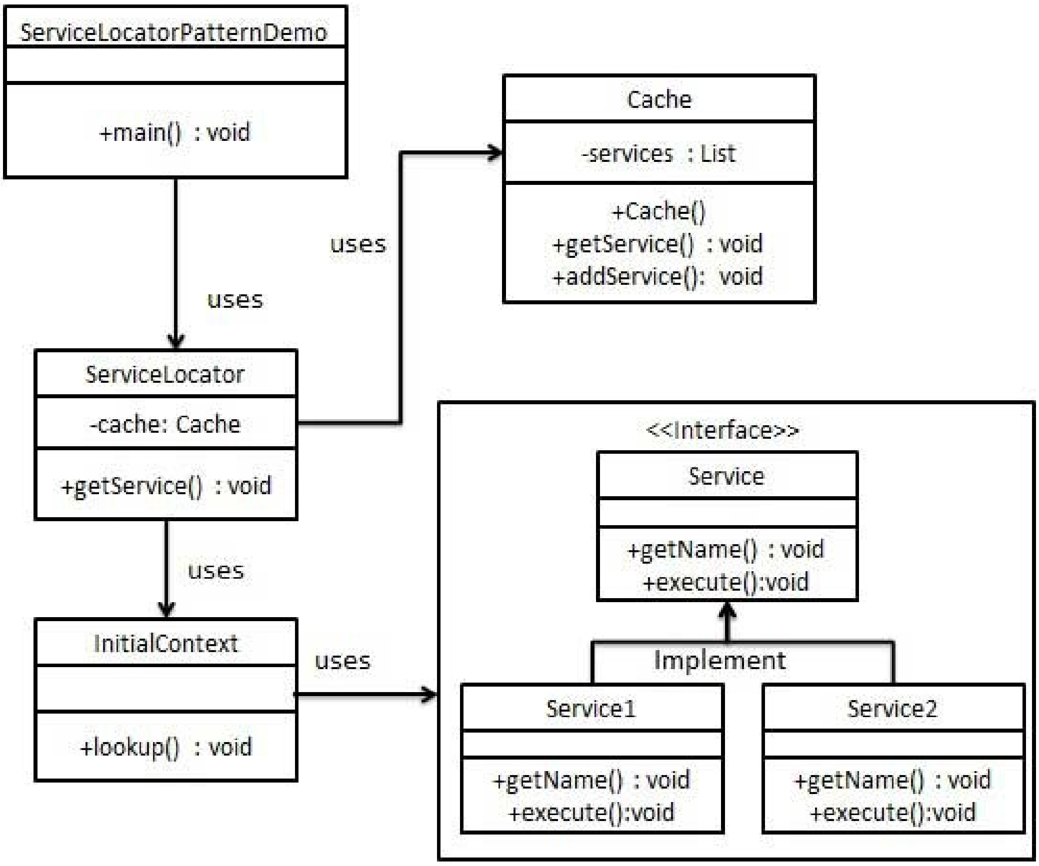
服务定位器模式（Service Locator Pattern）用在我们想使用 JNDI 查询定位各种服务的时候。考虑到为某个服务查找 JNDI 的代价很高，服务定位器模式充分利用了缓存技术。在首次请求某个服务时，服务定位器在 JNDI 中查找服务，并缓存该服务对象。当再次请求相同的服务时，服务定位器会在它的缓存中查找，这样可以在很大程度上提高应用程序的性能。以下是这种设计模式的实体。

- 服务（**Service**） - 实际处理请求的服务。对这种服务的引用可以在 JNDI 服务器中查找到。
- **Context /** 初始的 **Context** - JNDI Context 带有对要查找的服务的引用。
- 服务定位器（**Service Locator**） - 服务定位器是通过 JNDI 查找和缓存服务来获取服务的单点接触。
- 缓存（**Cache**） - 缓存存储服务的引用，以便复用它们。
- 客户端（**Client**） - Client 是通过 ServiceLocator 调用服务的对象。

实现

我们将创建 *ServiceLocator*、*InitialContext*、*Cache*、*Service* 作为表示实体的各种对象。*Service1* 和 *Service2* 表示实体服务。

ServiceLocatorPatternDemo，我们的演示类在这里是作为一个客户端，将使用 *ServiceLocator* 来演示服务定位器设计模式。



步骤 1

创建服务接口 Service。

Service.java

```
public interface Service {
   public String getName();
   public void execute();
}
```

步骤 2

创建实体服务。

Service1.java

```
public class Service1 implements Service {
   public void execute(){
      System.out.println("Executing Service1");
   }

   @Override
   public String getName() {
      return "Service1";
   }
}
```

Service2.java

```
public class Service2 implements Service {
   public void execute(){
      System.out.println("Executing Service2");
   }

   @Override
   public String getName() {
      return "Service2";
   }
}
```

步骤 3

为 JNDI 查询创建 InitialContext。

InitialContext.java

```
public class InitialContext {
   public Object lookup(String jndiName){
      if(jndiName.equalsIgnoreCase("SERVICE1")){
         System.out.println("Looking up and creating a new Service1 object");
         return new Service1();
      }else if (jndiName.equalsIgnoreCase("SERVICE2")){
         System.out.println("Looking up and creating a new Service2 object");
         return new Service2();
      }
      return null;
   }
}
```

步骤 4

创建缓存 Cache。

Cache.java

```
import java.util.ArrayList;
import java.util.List;

public class Cache {

   private List<Service> services;

   public Cache(){
      services = new ArrayList<Service>();
   }

   public Service getService(String serviceName){
      for (Service service : services) {
         if(service.getName().equalsIgnoreCase(serviceName)){
            System.out.println("Returning cached  "+serviceName+" object");
            return service;
         }
      }
      return null;
   }

   public void addService(Service newService){
      boolean exists = false;
      for (Service service : services) {
         if(service.getName().equalsIgnoreCase(newService.getName())){
            exists = true;
         }
      }
      if(!exists){
         services.add(newService);
      }
   }
}
```

步骤 5

创建服务定位器。

ServiceLocator.java

```
public class ServiceLocator {
   private static Cache cache;

   static {
      cache = new Cache();
   }

   public static Service getService(String jndiName){

      Service service = cache.getService(jndiName);

      if(service != null){
         return service;
      }

      InitialContext context = new InitialContext();
      Service service1 = (Service)context.lookup(jndiName);
      cache.addService(service1);
      return service1;
   }
}
```

步骤 6

使用 *ServiceLocator* 来演示服务定位器设计模式。

ServiceLocatorPatternDemo.java

```
public class ServiceLocatorPatternDemo {
   public static void main(String[] args) {
      Service service = ServiceLocator.getService("Service1");
      service.execute();
      service = ServiceLocator.getService("Service2");
      service.execute();
      service = ServiceLocator.getService("Service1");
      service.execute();
      service = ServiceLocator.getService("Service2");
      service.execute();
   }
}
```

步骤 7

验证输出。

```
Looking up and creating a new Service1 object
Executing Service1
Looking up and creating a new Service2 object
Executing Service2
Returning cached  Service1 object
Executing Service1
Returning cached  Service2 object
Executing Service2
```

传输对象模式

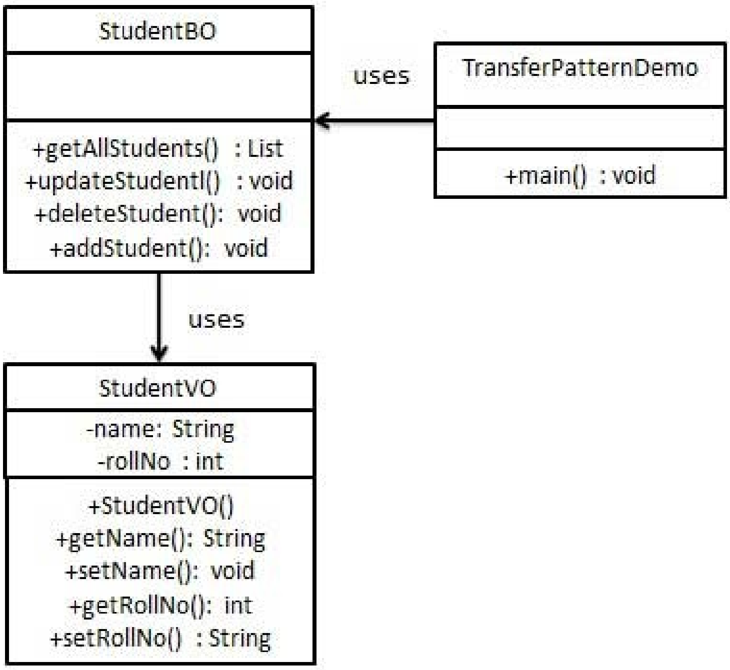
传输对象模式（Transfer Object Pattern）用于从客户端向服务器一次性传递带有多个属性的数据。传输对象也被称为数值对象。传输对象是一个具有 getter/setter 方法的简单的 POJO 类，它是可序列化的，所以它可以通过网络传输。它没有任何的行为。服务器端的业务类通常从数据库读取数据，然后填充 POJO，并把它发送到客户端或按值传递它。对于客户端，传输对象是只读的。客户端可以创建自己的传输对象，并把它传递给服务器，以便一次性更新数据库中的数值。以下是这种设计模式的实体。

- 业务对象（**Business Object**） - 为传输对象填充数据的业务服务。
- 传输对象（**Transfer Object**） - 简单的 POJO，只有设置/获取属性的方法。
- 客户端（**Client**） - 客户端可以发送请求或者发送传输对象到业务对象。

实现

我们将创建一个作为业务对象的 *StudentBO* 和作为传输对象的 *StudentVO*，它们都代表了我们的实体。

TransferObjectPatternDemo，我们的演示类在这里是作为一个客户端，将使用 *StudentBO* 和 *Student* 来演示传输对象设计模式。



步骤 1

创建传输对象。

StudentVO.java

```
public class StudentVO {
   private String name;
   private int rollNo;

   StudentVO(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getRollNo() {
      return rollNo;
   }

   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```

步骤 2

创建业务对象。

StudentBO.java

```
import java.util.ArrayList;
import java.util.List;

public class StudentBO {

   //列表是当作一个数据库
   List<StudentVO> students;

   public StudentBO(){
      students = new ArrayList<StudentVO>();
      StudentVO student1 = new StudentVO("Robert",0);
      StudentVO student2 = new StudentVO("John",1);
      students.add(student1);
      students.add(student2);
   }
   public void deleteStudent(StudentVO student) {
      students.remove(student.getRollNo());
      System.out.println("Student: Roll No "
      + student.getRollNo() +", deleted from database");
   }

   //从数据库中检索学生名单
   public List<StudentVO> getAllStudents() {
      return students;
   }

   public StudentVO getStudent(int rollNo) {
      return students.get(rollNo);
   }

   public void updateStudent(StudentVO student) {
      students.get(student.getRollNo()).setName(student.getName());
      System.out.println("Student: Roll No "
      + student.getRollNo() +", updated in the database");
   }
}
```

步骤 3

使用 *StudentBO* 来演示传输对象设计模式。

TransferObjectPatternDemo.java

```
public class TransferObjectPatternDemo {
   public static void main(String[] args) {
      StudentBO studentBusinessObject = new StudentBO();

      //输出所有的学生
      for (StudentVO student : studentBusinessObject.getAllStudents()) {
         System.out.println("Student: [RollNo : "
         +student.getRollNo()+", Name : "+student.getName()+" ]");
      }

      //更新学生
      StudentVO student =studentBusinessObject.getAllStudents().get(0);
      student.setName("Michael");
      studentBusinessObject.updateStudent(student);

      //获取学生
      studentBusinessObject.getStudent(0);
      System.out.println("Student: [RollNo : "
      +student.getRollNo()+", Name : "+student.getName()+" ]");
   }
}
```

步骤 4

验证输出。

```
Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]
```

W3School 正则表达式教程

来源：正则表达式教程

整理：飞龙

正则表达式 - 简介

除非您以前使用过正则表达式，否则您可能不熟悉此术语。但是，毫无疑问，您已经使用过不涉及脚本的某些正则表达式概念。

例如，您很可能使用 ? 和 通配符来查找硬盘上的文件。通配符匹配文件名中的单个字符，而 通配符匹配零个或多个字符。像 data?.dat 这样的模式将查找下列文件：

```
data1.dat
data2.dat
datax.dat
dataN.dat
```

使用 字符代替 ? 字符扩大了找到的文件的数量。*data*.dat 匹配下列所有文件：

```
data.dat
data1.dat
data2.dat
data12.dat
datax.dat
dataXYZ.dat
```

尽管这种搜索方法很有用，但它还是有限的。通过理解 * 通配符的工作原理，引入了正则表达式所依赖的概念，但正则表达式功能更强大，而且更加灵活。

正则表达式的使用，可以通过简单的办法来实现强大的功能。下面先给出一个简单的示例：

```
 ^.+@.+\\..+$
```

继续阅读本教程将让您也可以自由应用这样的代码。

为什么使用正则表达式？

典型的搜索和替换操作要求您提供与预期的搜索结果匹配的确切文本。虽然这种技术对于对静态文本执行简单搜索和替换任务可能已经足够了，但它缺乏灵活性，若采用这种方法搜索动态文本，即使不是不可能，至少也会变得很困难。

通过使用正则表达式，可以：

- 测试字符串内的模式。
  例如，可以测试输入字符串，以查看字符串内是否出现电话号码模式或信用卡号码模式。这称为数据验证。
- 替换文本。
  可以使用正则表达式来识别文档中的特定文本，完全删除该文本或者用其他文本替换

它。
- 基于模式匹配从字符串中提取子字符串。
  可以查找文档内或输入域内特定的文本。

例如，您可能需要搜索整个网站，删除过时的材料，以及替换某些 HTML 格式标记。在这种情况下，可以使用正则表达式来确定在每个文件中是否出现该材料或该 HTML 格式标记。此过程将受影响的文件列表缩小到包含需要删除或更改的材料的那些文件。然后可以使用正则表达式来删除过时的材料。最后，可以使用正则表达式来搜索和替换标记。

发展历史

正则表达式的"祖先"可以一直上溯至对人类神经系统如何工作的早期研究。Warren McCulloch 和 Walter Pitts 这两位神经生理学家研究出一种数学方式来描述这些神经网络。

1956 年, 一位叫 Stephen Kleene 的数学家在 McCulloch 和 Pitts 早期工作的基础上，发表了一篇标题为"神经网事件的表示法"的论文，引入了正则表达式的概念。正则表达式就是用来描述他称为"正则集的代数"的表达式，因此采用"正则表达式"这个术语。

随后，发现可以将这一工作应用于使用 Ken Thompson 的计算搜索算法的一些早期研究，Ken Thompson 是 Unix 的主要发明人。正则表达式的第一个实用应用程序就是 Unix 中的 qed 编辑器。

如他们所说，剩下的就是众所周知的历史了。从那时起直至现在正则表达式都是基于文本的编辑器和搜索工具中的一个重要部分。

应用领域

目前，正则表达式已经在很多软件中得到广泛的应用，包括 *nix（Linux, Unix等）、HP 等操作系统，PHP、C#、Java 等开发环境，以及很多的应用软件中，都可以看到正则表达式的影子。

C# 正则表达式

在我们的 C# 教程中，C# 正则表达式 这一章节专门介绍了有关 C# 正则表达式的知识。

Java 正则表达式

在我们的 Java 教程中，Java 正则表达式 这一章节专门介绍了有关 Java 正则表达式的知识。

JavaScript 正则表达式

在我们的 JavaScript 教程中，JavaScript RegExp 对象 这一章节专门介绍了有关 JavaScript 正则表达式的知识，同时我们还提供了完整的 JavaScript RegExp 对象参考手册。

Python 正则表达式

在我们的 Python 基础教程中，Python 正则表达式 这一章节专门介绍了有关 Python 正则表达式的知识。

Ruby 正则表达式

在我们的 Ruby 教程中，Ruby 正则表达式 这一章节专门介绍了有关 Ruby 正则表达式的知识。

正则表达式 - 语法

正则表达式(regular expression)描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。

- 列目录时， dir *.txt*或*ls* .txt中的*.txt*就不是一个正则表达式,因为这里与正则式的*的含义是不同的。
- 构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与运算符可以将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个的字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。

正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为"元字符"）组成的文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

普通字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符。这包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

非打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

字符	描述
\cx	匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

特殊字符

所谓特殊字符，就是一些有特殊含义的字符，如上面说的".*txt*"中的，简单的说就是表示任何字符串的意思。如果要查找文件名中有的文件，则需要对进行转义，即在其前加一个\。ls *.txt。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符"转义"，即，将反斜杠字符 () 放在它们前面。下表列出了正则表达式中的特殊字符：

特别字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 $。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 (和)。
*	匹配前面的子表达式零次或多次。要匹配 字符，请使用 \。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 +。
.	匹配除换行符 \n之外的任何单字符。要匹配 .，请使用 \。
[	标记一个中括号表达式的开始。要匹配 [，请使用 [。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\' 匹配 "\"，而 '(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 {。
|	指明两项之间的一个选择。要匹配 |，请使用 \ |。

限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有*或+或?或{n}或{n,}或{n,m}共6种。

正则表达式的限定符有：

字符	描述
*	匹配前面的子表达式零次或多次。例如，zo 能匹配 *"z"* 以及 *"zoo"*。 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

由于章节编号在大的输入文档中会很可能超过九，所以您需要一种方式来处理两位或三位章节编号。限定符给您这种能力。下面的正则表达式匹配编号为任何位数的章节标题：

```
/Chapter [1-9][0-9]*/
```

请注意，限定符出现在范围表达式之后。因此，它应用于整个范围表达式，在本例中，只指定从 0 到 9 的数字（包括 0 和 9）。

这里不使用 + 限定符，因为在第二个位置或后面的位置不一定需要有一个数字。也不使用？字符，因为它将章节编号限制到只有两位数。您需要至少匹配 Chapter 和空格字符后面的一个数字。

如果您知道章节编号被限制为只有 99 章，可以使用下面的表达式来至少指定一位但至多两位数字。

```
/Chapter [0-9]{1,2}/
```

上面的表达式的缺点是，大于 99 的章节编号仍只匹配开头两位数字。另一个缺点是 Chapter 0 也将匹配。只匹配两位数字的更好的表达式如下：

```
/Chapter [1-9][0-9]?/
```

或

```
/Chapter [1-9][0-9]{0,1}/
```

***、+和?**限定符都是贪婪的，因为它们会尽可能多的匹配文字，只有在它们的后面加上一个**?**就可以实现非贪婪或最小匹配。

例如，您可能搜索 HTML 文档，以查找括在 H1 标记内的章节标题。该文本在您的文档中如下：

```
<H1>Chapter 1 – Introduction to Regular Expressions</H1>
```

下面的表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。

```
/<.*>/
```

如果您只需要匹配开始 H1 标记，下面的"非贪心"表达式只匹配 <H1>。

```
/<.*?>/
```

通过在 *、+ 或 ? 限定符之后放置 ?，该表达式从"贪心"表达式转换为"非贪心"表达式或者最小匹配。

定位符

定位符使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。

定位符用来描述字符串或单词的边界，^和$分别指字符串的开始与结束，\b描述单词的前或后边界，\B表示非单词边界。

正则表达式的限定符有：

字符	描述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个字边界，即字与空格间的位置。
\B	非字边界匹配。

注意：不能将限定符与定位点一起使用。由于在紧靠换行或者字边界的前面或后面不能有一个以上位置，因此不允许诸如 ^* 之类的表达式。

若要匹配一行文本开始处的文本，请在正则表达式的开始使用 ^ 字符。不要将 ^ 的这种用法与中括号表达式内的用法混淆。

若要匹配一行文本的结束处的文本，请在正则表达式的结束处使用 $ 字符。

若要在搜索章节标题时使用定位点，下面的正则表达式匹配一个章节标题，该标题只包含两个尾随数字，并且出现在行首：

```
/^Chapter [1-9][0-9]{0,1}/
```

真正的章节标题不仅出现行的开始处，而且它还是该行中仅有的文本。它即出现在行首又出现在同一行的结尾。下面的表达式能确保指定的匹配只匹配章节而不匹配交叉引用。通过创建只匹配一行文本的开始和结尾的正则表达式，就可做到这一点。

```
/^Chapter [1-9][0-9]{0,1}$/
```

匹配字边界稍有不同，但向正则表达式添加了很重要的能力。字边界是单词和空格之间的位置。非字边界是任何其他位置。下面的表达式匹配单词 Chapter 的开头三个字符，因为这三个字符出现字边界后面：

```
/\bCha/
```

\b 字符的位置是非常重要的。如果它位于要匹配的字符串的开始，它在单词的开始处查找匹配项。如果它位于字符串的结尾，它在单词的结尾处查找匹配项。例如，下面的表达式匹配单词 Chapter 中的字符串 ter，因为它出现在字边界的前面：

```
/ter\b/
```

下面的表达式匹配 Chapter 中的字符串 apt，但不匹配 aptitude 中的字符串 apt：

```
/\Bapt/
```

字符串 apt 出现在单词 Chapter 中的非字边界处，但出现在单词 aptitude 中的字边界处。对于 \B 非字边界运算符，位置并不重要，因为匹配不关心究竟是单词的开头还是结尾。

选择

用圆括号将所有选择项括起来，相邻的选择项之间用|分隔。但用圆括号会有一个副作用，是相关的匹配会被缓存，此时可用?:放在第一个选项前来消除这种副作用。

其中?:是非捕获元之一，还有两个非捕获元是?=和?!，这两个还有更多的含义，前者为正向预查，在任何开始匹配圆括号内的正则表达式模式的位置来匹配搜索字符串，后者为负向预查，在任何开始不匹配该正则表达式模式的位置来匹配搜索字符串。

反向引用

对一个正则表达式模式或部分模式两边添加圆括号将导致相关匹配存储到一个临时缓冲区中，所捕获的每个子匹配都按照在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始，最多可存储 99 个捕获的子表达式。每个缓冲区都可以使用 '\n' 访问，其中 n 为一个标识特定缓冲区的一位或两位十进制数。

可以使用非捕获元字符 '?:'、'?=' 或 '?!' 来重写捕获，忽略对相关匹配的保存。

反向引用的最简单的、最有用的应用之一，是提供查找文本中两个相同的相邻单词的匹配项的能力。以下面的句子为例：

```
Is is the cost of of gasoline going up up?
```

上面的句子很显然有多个重复的单词。如果能设计一种方法定位该句子，而不必查找每个单词的重复出现，那该有多好。下面的正则表达式使用单个子表达式来实现这一点：

```
/\b([a-z]+) \1\b/gi
```

捕获的表达式，正如 [a-z]+ 指定的，包括一个或多个字母。正则表达式的第二部分是对以前捕获的子匹配项的引用，即，单词的第二个匹配项正好由括号表达式匹配。\1 指定第一个子匹配项。字边界元字符确保只检测整个单词。否则，诸如"is issued"或"this is"之类的词组将不能正确地被此表达式识别。

正则表达式后面的全局标记 (g) 指示，将该表达式应用到输入字符串中能够查找到的尽可能多的匹配。表达式的结尾处的不区分大小写 (i) 标记指定不区分大小写。多行标记指定换行符的两边可能出现潜在的匹配。

反向引用还可以将通用资源指示符 (URI) 分解为其组件。假定您想将下面的 URI 分解为协议（ftp、http 等等）、域地址和页/路径：

```
http://www.w3cschool.cc:80/html/html-tutorial.html
```

下面的正则表达式提供该功能：

```
/(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)/
```

第一个括号子表达式捕获 Web 地址的协议部分。该子表达式匹配在冒号和两个正斜杠前面的任何单词。第二个括号子表达式捕获地址的域地址部分。子表达式匹配 / 或 : 之外的一个或多个字符。第三个括号子表达式捕获端口号（如果指定了的话）。该子表达式匹配冒号后面的零个或多个数字。只能重复一次该子表达式。最后，第四个括号子表达式捕获 Web 地址指定的路径和/或页信息。该子表达式能匹配不包括 # 或空格字符的任何字符序列。

将正则表达式应用到上面的 URI，各子匹配项包含下面的内容：

- 第一个括号子表达式包含"http"
- 第二个括号子表达式包含"www.w3cschool.cc"
- 第三个括号子表达式包含":80"
- 第四个括号子表达式包含"/html/html-tutorial.html"

正则表达式 - 元字符

下表包含了元字符的完整列表以及它们在正则表达式上下文中的行为：

字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个 向后引用、或一个八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 "\(" 则匹配 "("。
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。
$	匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。
*	匹配前面的子表达式零次或多次。例如，zo 能匹配 *"z"* 以及 *"zoo"*。 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如， "do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时， 匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如， 对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
(pattern)	匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在VBScript 中使用 SubMatches 集合，在JScript 中则使用 $0…$9 属性。要匹配圆括号字符，请使用 '\(' 或 '\)'。
(?:pattern)	匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用 "或" 字符 (|) 来组合一个模式的各个部分是很有用。例如， 'industr(?:y|ies) 就是一个比 'industry|industries' 更简略的表达式。
	正向预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个

(? =pattern)	(?=95|98|NT|2000)' 能匹配 "Windows 2000" 中的 "Windows" ，但不能匹配 "Windows 3.1" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?!pattern)	负向预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如'Windows (?!95|98|NT|2000)' 能匹配 "Windows 3.1" 中的 "Windows"，但不能匹配 "Windows 2000" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
x|y	匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。
[xyz]	字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。
[^xyz]	负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\cx	匹配由 x 指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 [^0-9]。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'。
\W	匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'。
	匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数

\xn	字长。例如，'\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。
\num	匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。例如，'(.)\1' 匹配两个连续的相同字符。
\n	标识一个八进制转义值或一个向后引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。
\nm	标识一个八进制转义值或一个向后引用。如果 \nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果 \nm 之前至少有 n 个获取，则 n 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字 (0-7)，则 \nm 将匹配八进制转义值 nm。
\nml	如果 n 为八进制数字 (0-3)，且 m 和 l 均为八进制数字 (0-7)，则匹配八进制转义值 nml。
\un	匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (?)。

正则表达式 - 运算符优先级

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。

相同优先级的从左到右进行运算，不同优先级的运算先高后低。下表从最高到最低说明了各种正则表达式运算符的优先级顺序：

运算符	描述
\	转义符
(), (?:), (?=), []	圆括号和方括号
*, +, ?, {n}, {n,}, {n,m}	限定符
^, $, \任何元字符、任何字符	定位点和序列（即：位置和顺序）
|	替换，"或"操作 字符具有高于替换运算符的优先级，使得"m|food"匹配"m"或"food"。若要匹配"mood"或"food"，请使用括号创建子表达式，从而产生"(m|f)ood"。

正则表达式 - 匹配规则

基本模式匹配

一切从最基本的开始。模式，是正规表达式最基本的元素，它们是一组描述字符串特征的字符。模式可以很简单，由普通的字符串组成，也可以非常复杂，往往用特殊的字符表示一个范围内的字符、重复出现，或表示上下文。例如：

```
^once
```

这个模式包含一个特殊的字符^，表示该模式只匹配那些以once开头的字符串。例如该模式与字符串"once upon a time"匹配，与"There once was a man from NewYork"不匹配。正如如^符号表示开头一样，$符号用来匹配那些以给定模式结尾的字符串。

```
bucket$
```

这个模式与"Who kept all of this cash in a bucket"匹配，与"buckets"不匹配。字符^和$同时使用时，表示精确匹配（字符串与模式一样）。例如：

```
^bucket$
```

只匹配字符串"bucket"。如果一个模式不包括^和$，那么它与任何包含该模式的字符串匹配。例如：模式

```
once
```

与字符串

```
There once was a man from NewYork
Who kept all of his cash in a bucket.
```

是匹配的。

在该模式中的字母(o-n-c-e)是字面的字符，也就是说，他们表示该字母本身，数字也是一样的。其他一些稍微复杂的字符，如标点符号和白字符（空格、制表符等），要用到转义序列。所有的转义序列都用反斜杠()打头。制表符的转义序列是：\t。所以如果我们要检测一个字符串是否以制表符开头，可以用这个模式：

```
^\t
```

类似的，用\n表示"新行"，\r表示回车。其他的特殊符号，可以用在前面加上反斜杠，如反斜杠本身用\表示，句号.用.表示，以此类推。

字符簇

在INTERNET的程序中，正规表达式通常用来验证用户的输入。当用户提交一个FORM以后，要判断输入的电话号码、地址、EMAIL地址、信用卡号码等是否有效，用普通的基于字面的字符是不够的。

所以要用一种更自由的描述我们要的模式的办法，它就是字符簇。要建立一个表示所有元音字符的字符簇，就把所有的元音字符放在一个方括号里：

```
[AaEeIiOoUu]
```

这个模式与任何元音字符匹配，但只能表示一个字符。用连字号可以表示一个字符的范围，如：

```
[a-z] //匹配所有的小写字母
[A-Z] //匹配所有的大写字母
[a-zA-Z] //匹配所有的字母
[0-9] //匹配所有的数字
[0-9\.\-] //匹配所有的数字，句号和减号
[ \f\r\t\n] //匹配所有的白字符
```

同样的，这些也只表示一个字符，这是一个非常重要的。如果要匹配一个由一个小写字母和一位数字组成的字符串，比如"z2"、"t6"或"g7"，但不是"ab2"、"r2d3" 或"b52"的话，用这个模式：

```
^[a-z][0-9]$
```

尽管[a-z]代表26个字母的范围，但在这里它只能与第一个字符是小写字母的字符串匹配。

前面曾经提到^表示字符串的开头，但它还有另外一个含义。当在一组方括号里使用^是，它表示"非"或"排除"的意思，常常用来剔除某个字符。还用前面的例子，我们要求第一个字符不能是数字：

```
^[^0-9][0-9]$
```

这个模式与"&5"、"g7"及"-2"是匹配的，但与"12"、"66"是不匹配的。下面是几个排除特定字符的例子：

```
[^a-z] //除了小写字母以外的所有字符
[^\\\/\^] //除了(\)(/)(^)之外的所有字符
[^\"\'] //除了双引号(")和单引号(')之外的所有字符
```

特殊字符"." (点，句号)在正规表达式中用来表示除了"新行"之外的所有字符。所以模式"^.5$"与任何两个字符的、以数字5结尾和以其他非"新行"字符开头的字符串匹配。模式"."可以匹配任何字符串，除了空串和只包括一个"新行"的字符串。

PHP的正规表达式有一些内置的通用字符簇，列表如下：

字符簇	描述
[[:alpha:]]	任何字母
[[:digit:]]	任何数字
[[:alnum:]]	任何字母和数字
[[:space:]]	任何空白字符
[[:upper:]]	任何大写字母
[[:lower:]]	任何小写字母
[[:punct:]]	任何标点符号
[[:xdigit:]]	任何16进制的数字，相当于[0-9a-fA-F]

确定重复出现

到现在为止，你已经知道如何去匹配一个字母或数字，但更多的情况下，可能要匹配一个单词或一组数字。一个单词有若干个字母组成，一组数字有若干个单数组成。跟在字符或字符簇后面的花括号({})用来确定前面的内容的重复出现的次数。

字符簇	描述
^[a-zA-Z_]$	所有的字母和下划线
^[[:alpha:]]{3}$	所有的3个字母的单词
^a$	字母a
^a{4}$	aaaa
^a{2,4}$	aa,aaa或aaaa
^a{1,3}$	a,aa或aaa
^a{2,}$	包含多于两个a的字符串
^a{2,}	如：aardvark和aaab，但apple不行
a{2,}	如：baad和aaa，但Nantucket不行
\t{2}	两个制表符
.{2}	所有的两个字符

这些例子描述了花括号的三种不同的用法。一个数字，{x}的意思是"前面的字符或字符簇只出现x次"；一个数字加逗号，{x,}的意思是"前面的内容出现x或更多的次数"；两个用逗号分隔的数字，{x,y}表示"前面的内容至少出现x次，但不超过y次"。我们可以把模式扩展到更多的单词或数字：

```
^[a-zA-Z0-9_]{1,}$ //所有包含一个以上的字母、数字或下划线的字符串
^[0-9]{1,}$ //所有的正数
^\-{0,1}[0-9]{1,}$ //所有的整数
^\-{0,1}[0-9]{0,}\.{0,1}[0-9]{0,}$ //所有的小数
```

最后一个例子不太好理解，是吗？这么看吧：与所有以一个可选的负号(-{0,1})开头(^)、跟着0个或更多的数字([0-9]{0,})、和一个可选的小数点(.{0,1})再跟上0个或多个数字([0-9]{0,})，并且没有其他任何东西($)。下面你将知道能够使用的更为简单的方法。

特殊字符"?"与{0,1}是相等的，它们都代表着："0个或1个前面的内容"或"前面的内容是可选的"。所以刚才的例子可以简化为：

```
^\-?[0-9]{0,}\.?[0-9]{0,}$
```

特殊字符"*"与{0,}是相等的，它们都代表着"0个或多个前面的内容"。最后，字符"+"与 {1,}是相等的，表示"1个或多个前面的内容"，所以上面的4个例子可以写成：

```
^[a-zA-Z0-9_]+$ //所有包含一个以上的字母、数字或下划线的字符串
^[0-9]+$ //所有的正数
^\-?[0-9]+$ //所有的整数
^\-?[0-9]*\.?[0-9]*$ //所有的小数
```

当然这并不能从技术上降低正规表达式的复杂性，但可以使它们更容易阅读。

正则表达式 - 示例

简单表达式

正则表达式的最简单形式是在搜索字符串中匹配其本身的单个普通字符。例如，单字符模式，如 A，不论出现在搜索字符串中的何处，它总是匹配字母 A。下面是一些单字符正则表达式模式的示例：

```
/a/
/7/
/M/
```

可以将许多单字符组合起来以形成大的表达式。例如，以下正则表达式组合了单字符表达式：a、7 和 M。

```
/a7M/
```

请注意，没有串联运算符。只须在一个字符后面键入另一个字符。

字符匹配

句点 (.) 匹配字符串中的各种打印或非打印字符，只有一个字符例外。这个例外就是换行符 (\n)。下面的正则表达式匹配 aac、abc、acc、adc 等等，以及 a1c、a2c、a-c 和 a#c：

```
/a.c/
```

若要匹配包含文件名的字符串，而句点 (.) 是输入字符串的组成部分，请在正则表达式中的句点前面加反斜扛 () 字符。举例来说明，下面的正则表达式匹配 filename.ext：

```
/filename\.ext/
```

这些表达式只让您匹配"任何"单个字符。可能需要匹配列表中的特定字符组。例如，可能需要查找用数字表示的章节标题（Chapter 1、Chapter 2 等等）。

中括号表达式

若要创建匹配字符组的一个列表，请在方括号（[和]）内放置一个或更多单个字符。当字符括在中括号内时，该列表称为"中括号表达式"。与在任何别的位置一样，普通字符在中括号内表示其本身，即，它在输入文本中匹配一次其本身。大多数特殊字符在中括号表达式内出现时失去它们的意义。不过也有一些例外，如：

- 如果] 字符不是第一项，它结束一个列表。若要匹配列表中的] 字符，请将它放在第一位，紧跟在开始 [后面。
- \ 字符继续作为转义符。若要匹配 \ 字符，请使用 \\。

括在中括号表达式中的字符只匹配处于正则表达式中该位置的单个字符。以下正则表达式匹配 Chapter 1、Chapter 2、Chapter 3、Chapter 4 和 Chapter 5：

```
/Chapter [12345]/
```

请注意，单词 Chapter 和后面的空格的位置相对于中括号内的字符是固定的。中括号表达式指定的只是匹配紧跟在单词 Chapter 和空格后面的单个字符位置的字符集。这是第九个字符位置。

若要使用范围代替字符本身来表示匹配字符组，请使用连字符 (-) 将范围中的开始字符和结束字符分开。单个字符的字符值确定范围内的相对顺序。下面的正则表达式包含范围表达式，该范围表达式等效于上面显示的中括号中的列表。

```
/Chapter [1-5]/
```

当以这种方式指定范围时，开始值和结束值两者都包括在范围内。注意，还有一点很重要，按 Unicode 排序顺序，开始值必须在结束值的前面。

若要在中括号表达式中包括连字符，请采用下列方法之一：

- 用反斜杠将它转义：

  ```
  [\-]
  ```

- 将连字符放在中括号列表的开始或结尾。下面的表达式匹配所有小写字母和连字符：

  ```
  [-a-z]
  [a-z-]
  ```

- 创建一个范围，在该范围中，开始字符值小于连字符，而结束字符值等于或大于连字符。下面的两个正则表达式都满足这一要求：

  ```
  [!--]
  [!-~]
  ```

若要查找不在列表或范围内的所有字符，请将插入符号 (^) 放在列表的开头。如果插入字符出现在列表中的其他任何位置，则它匹配其本身。下面的正则表达式匹配编号大于 5 的章节标题：

```
/Chapter [^12345]/
```

在上面的示例中，表达式在第九个位置匹配 1、2、3、4 或 5 之外的任何数字字符。这样，例如，Chapter 7 就是一个匹配项，Chapter 9 也是一个匹配项。

上面的表达式可以使用连字符 (-) 来表示：

```
/Chapter [^1-5]/
```

中括号表达式的典型用途是指定任何大写或小写字母或任何数字的匹配。下面的表达式指定这样的匹配：

```
/[A-Za-z0-9]/
```

替换和分组

替换使用 | 字符来允许在两个或多个替换选项之间进行选择。例如，可以扩展章节标题正则表达式，以返回比章标题范围更广的匹配项。但是，这并不象您可能认为的那样简单。替换匹配 | 字符任一侧最大的表达式。

您可能认为，下面的表达式匹配出现在行首和行尾、后面跟一个或两个数字的 Chapter 或 Section：

```
/^Chapter|Section [1-9][0-9]{0,1}$/
```

很遗憾，上面的正则表达式要么匹配行首的单词 Chapter，要么匹配行尾的单词 Section 及跟在其后的任何数字。如果输入字符串是 Chapter 22，那么上面的表达式只匹配单词 Chapter。如果输入字符串是 Section 22，那么该表达式匹配 Section 22。

若要使正则表达式更易于控制，可以使用括号来限制替换的范围，即，确保它只应用于两个单词 Chapter 和 Section。但是，括号也用于创建子表达式，并可能捕获它们以供以后使用，这一点在有关反向引用的那一节讲述。通过在上面的正则表达式的适当位置添加括号，就可以使该正则表达式匹配 Chapter 1 或 Section 3。

下面的正则表达式使用括号来组合 Chapter 和 Section，以便表达式正确地起作用：

```
/^(Chapter|Section) [1-9][0-9]{0,1}$/
```

尽管这些表达式正常工作，但 Chapter|Section 周围的括号还将捕获两个匹配字中的任一个供以后使用。由于在上面的表达式中只有一组括号，因此，只有一个被捕获的"子匹配项"。

在上面的示例中，您只需要使用括号来组合单词 Chapter 和 Section 之间的选择。若要防止匹配被保存以备将来使用，请在括号内正则表达式模式之前放置 ?:。下面的修改提供相同的能力而不保存子匹配项：

```
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/
```

除 ?: 元字符外，两个其他非捕获元字符创建被称为"预测先行"匹配的某些内容。正向预测先行使用 ?= 指定，它匹配处于括号中匹配正则表达式模式的起始点的搜索字符串。反向预测先行使用 ?! 指定，它匹配处于与正则表达式模式不匹配的字符串的起始点的搜索字符串。

例如，假设您有一个文档，该文档包含指向 Windows 3.1、Windows 95、Windows 98 和 Windows NT 的引用。再进一步假设，您需要更新该文档，将指向 Windows 95、Windows 98 和 Windows NT 的所有引用更改为 Windows 2000。下面的正则表达式（这是一个正向预测先行的示例）匹配 Windows 95、Windows 98 和 Windows NT：

```
/Windows(?=95 |98 |NT )/
```

找到一处匹配后，紧接着就在匹配的文本（不包括预测先行中的字符）之后搜索下一处匹配。例如，如果上面的表达式匹配 Windows 98，将在 Windows 之后而不是在 98 之后继续搜索。

其他示例

下面列出一些正则表达式示例：

<table>
<tr><th>正则表达式</th><th>描述</th></tr>
<tr><td>/\b([a-z]+) \1\b/gi</td><td>一个单词连续出现的位置。</td></tr>
<tr><td>/(\w+):\/\/([^/:]+)(:\d)?([^#])/</td><td>将一个URL解析为协议、域、端口及相对路径。</td></tr>
<tr><td>/^(?:Chapter|Section) [1-9][0-9]{0,1}$/</td><td>定位章节的位置。</td></tr>
<tr><td>/[-a-z]/</td><td>A至z共26个字母再加一个-号。</td></tr>
<tr><td>/ter\b/</td><td>可匹配chapter，而不能匹配terminal。</td></tr>
<tr><td>/\Bapt/</td><td>可匹配chapter，而不能匹配aptitude。</td></tr>
<tr><td>/Windows(?=95 |98 |NT)/</td><td>可匹配Windows95或Windows98或WindowsNT，当找到一个匹配后，从Windows后面开始进行下一次的检索匹配。</td></tr>
<tr><td>/^\s*$/</td><td>匹配空行。</td></tr>
<tr><td>/\d{2}-\d{5}/</td><td>验证由两位数字、一个连字符再加 5 位数字组成的 ID 号。</td></tr>
<tr><td>/<\s*(\S+)(\s[^>]*)?>[\s\S]*<\s*\/\1\s*>/</td><td>匹配 HTML 标记。</td></tr>
</table>

免责声明

版权信息

菜鸟教程（www.runoob.com）刊载的所有内容，包括文字、图片、音频、视频、软件、程序、以及网页版式设计等均在网上搜集。

菜鸟教程提供的内容仅用于个人学习、研究或欣赏。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关

访问者可将本网站提供的内容或服务用于个人学习、研究或欣赏，以及其他非商业性或非盈利性用途，但同时应遵守著作权法及其他相关法律的规定，不得侵犯本网站及相关权利人的合法权利。

本网站内容原作者如不愿意在本网站刊登内容，请及时通知本站，予以删除。

本网站的编程技术内容大部分翻译自 W3Schools Online Web Tutorials 与 Tutorialspoint, 内容原作者如不愿意在本网站刊登内容，请及时通知本站，予以删除。

链接到菜鸟教程

任何网站都可以链接到菜鸟教程的任何页面。

如果您需要在对少量内容进行引用，请务必在引用该内容的页面添加指向被引用页面的链接。

保证

菜鸟教程不提供任何形式的保证。所有与使用本站相关的直接风险均由用户承担。菜鸟教程提供的所有代码均为实例，并不对性能、适用性、适销性或/及其他方面提供任何保证。

菜鸟教程的内容可能包含不准确性或错误。菜鸟教程不对本网站及其内容的准确性进行保证。如果您发现本站点及其内容包含错误，请联系我们以便这些错误得到及时的更正：429240967@qq.com

您的行为

当您使用本站点时，说明您已经同意并接受本页面的所有信息。

联系方式

联系邮箱：429240967@qq.com（相关事务请发函至该邮箱）